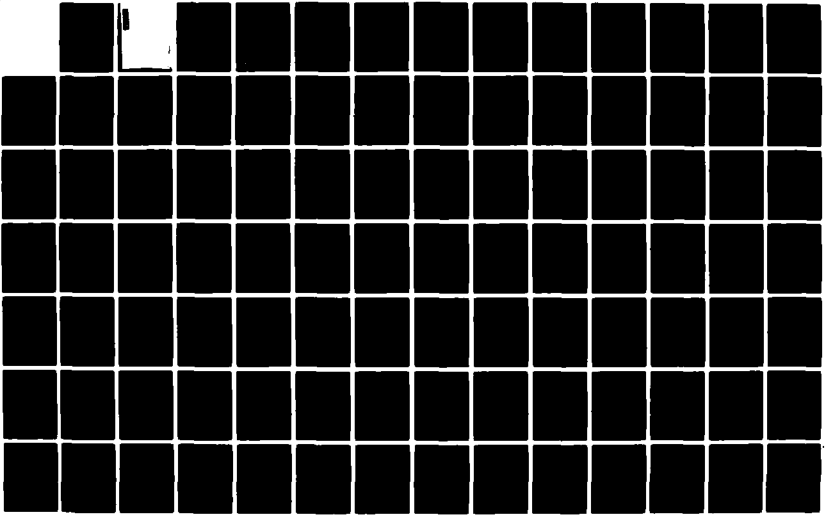
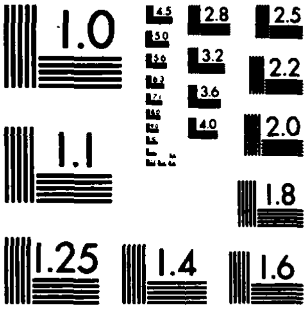
AD-A123 943 PERFORMANCE MODELS FOR MULTIPROCESSOR COMPUTER SYSTEMS 1/2
(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB
D W YEN OCT 80 R-894 N00014-79-C-0424

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ADA123943

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.
AD-A123943

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
PERFORMANCE MODELS FOR MULTIPROCESSOR COMPUTER SYSTEMS

5. TYPE OF REPORT & PERIOD COVERED
Technical Panel

6. PERFORMING ORG. REPORT NUMBER
R-894; UILU-ENG 80-2226

7. AUTHOR(s)
David Wei-Luen Yen

8. CONTRACT OR GRANT NUMBER(s)
N00014-79-C-0424
F33615-78-C-1559

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Joint Services Electronics Program

12. REPORT DATE
October, 1980

13. NUMBER OF PAGES
178

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
multiprocessors, interleaved memories,
memory bandwidth, performance evaluation,
shared resource models

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Multiprocessing is an effective architectural approach to enhance the performance ot computer systems. However, various problems involved in multiprocessing may severely degrade system performance.

This research has mainly centered on the memory interference problem in tightly coupled multiprocessor computer systems. Depending on the nature of the memory-requesting mechanism, discussion is centered on two important cases of such systems.

20. ABSTRACT (continued)

The memory interference in multiprocessor systems with time-division-multiplexed (TDM) busses is first discussed. The discussion starts from Emer's model for a multiple-instruction-stream pipelined processor with a single fixed-cycle shared resource. Generalizations of that model for systems with multiple resources, resources having more general cycle times, and/or unassigned time slots, are discussed. Provisions for the application of the model to programs with critical sections treated as software resources are also covered. Measured performance data from the execution of matrix multiplication on a local multiprocessor system is used to check the above model. Two other models for matrix multiplication execution are also presented for comparison. These two models model the imperfect job sharing among processors at the end of a computation, which has not been previously modeled.

A general model for the memory interference in synchronous multiprocessor systems which allow arbitrary memory request rates, non-uniform memory references, and unequal processor priorities is presented next. For the case of uniform memory access, an improved model based on a steady flow concept is discussed. With the aid of simulation results, this model is compared to other models in the entire range of memory request rate ((0,1]) to demonstrate its accuracy. This model is further shown to be extendable to deal with multiprocessor systems where different memory service priorities are associated with different processor categories.

Several application examples which make use of the memory interference models derived are presented. First, an algorithm is proposed for the estimation of the execution time of a program running in a multiprocessor system. Such an algorithm can be used to pick a computation decomposition which best utilizes the available computing power. A case study of the effect of computation decomposition on the performance of Gaussian Elimination is presented. The execution of matrix multiplication in a multiprocessor system with virtual memory was evaluated by simulation, in which a memory interference model capable of dealing with priority was used to dynamically modify various job execution times according to the number of processors and I/O channels active in the system.

DTIC
COPY
INSPECTED
2

Accession For
NTIS GRA&I ☑
DTIC TAB □
Unannounced □
Justification
By
Distribution/
Availability Codes
Avail and/or
Dist Special
A

UILU-ENG 80-2226

PERFORMANCE MODELS FOR MULTIPROCESSOR COMPUTER SYSTEMS

by

David Wei-Luen Yen

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract N00014-79-C-0424 and in part by the Wright Patterson Avionics Laboratory under Contract F33615-78-C-1559.

Reproduction in whole or in part is permitted for any purpose of the United States Government.

Approved for public release. Distribution unlimited.

PERFORMANCE MODELS FOR MULTIPROCESSOR COMPUTER SYSTEMS

BY

DAVID WEI-LUEN YEN

B.S., National Taiwan University, 1973
M.S., University of Illinois, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1980

Urbana, Illinois

ACKNOWLEDGMENT

The author wishes to express his sincere appreciation to his thesis advisor Professor Edward S. Davidson for his great contributions throughout the course of this work. Dr. Davidson´s patient guidance, acute insight, and continual encouragement have been invaluable to the progress of this research, while his genuine personal warmth, kind concern, and true care have been an indispensable support for the author during the period of his study.

The author would also like to thank Professor Janak H. Patel for his stimulating advice, fruitful discussions and friendship. Thanks are also due to Professor Michael Schlansker, B. R. Rau, Richard Brown, and Jacob Abraham for their valuable advice and friendship.

The author thanks his friends and colleagues, Tim Chou, Alan Gant, Larry Hanes, and Phil Yeh for helpful discussions and various assistances.

The author owes special gratitude to his wife, Grace Shau-Ling, for her patience and love. The author would also like to thank his brother, Wei-chen, for his valuable discussions and contribution to this dissertation.

Finally, the author wishes to thank Trudy Little for the typing of formulas and figure legends of this dissertation.

TABLE OF CONTENTS

PERFORMANCE MODELS FOR MULTIPROCESSOR COMPUTER SYSTEMS

David Wei-Luen Yen, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1980.

Multiprocessing is an effective architectural approach to enhance the performance of computer systems. However, various problems involved in multiprocessing may severely degrade system performance.

This research has mainly centered on the memory interference problem in tightly coupled multiprocessor computer systems. Depending on the nature of the memory-requesting mechanism, discussion is centered on two important cases of such systems.

The memory interference in multiprocessor systems with time-division- multiplexed (TDM) busses is first discussed. The discussion starts from Emer's model for a multiple-instruction-stream pipelined processor with a single fixed-cycle shared resource. Generalizations of that model for systems with multiple resources, resources having more general cycle times, and/or unassigned time slots, are discussed. Provisions for the application of the model to programs with critical sections treated as software resources are also covered. Measured performance data from the execution of matrix multiplication on a local multiprocessor system is used to check the above model. Two other models for matrix multiplication execution are also presented for

comparison. These two models model the imperfect job sharing among processors at the end of a computation, which has not been previously modeled.

A general model for the memory interference in synchronous multiprocessor systems which allow arbitrary memory request rates, non-uniform memory references, and unequal processor priorities is presented next. For the case of uniform memory access, an improved model based on a steady flow concept is discussed. With the aid of simulation results, this model is compared to other models in the entire range of memory request rate ($(0,1]$) to demonstrate its accuracy. This model is further shown to be extendable to deal with multiprocessor systems where different memory service priorities are associated with different processor categories.

Several application examples which make use of the memory interference models derived are presented. First, an algorithm is proposed for the estimation of the execution time of a program running in a multiprocessor system. Such an algorithm can be used to pick a computation decomposition which best utilizes the available computing power. A case study of the effect of computation decomposition on the performance of Gaussian Elimination is presented. The execution of matrix multiplication in a multiprocessor system with virtual memory was evaluated by simulation, in which a memory interference model capable of dealing with priority was used to dynamically modify various job execution times according to the number of processors and I/O channels active in the system.

LIST OF FIGURES

Figure Page

Figure Page

TABLES

CHAPTER 1

Introduction

1.1 Problem Statement and An Overview of the Thesis

Very often in our lives we see tasks that are considered too big for a single person to handle. Unless someone with extraordinary capability is readily available, a team of people, perhaps with differing specialties, is required to accomplish it. Similarly, as more sophisticated and ambitious computer applications are attempted, it is often difficult to build a single-processor computer system which is powerful enough to handle the problem. Advances in technology have so far been driving computer performance upward. Although technology will continue to improve the performance of computer systems, people can no longer solely count on it since it has begun to approach certain physical limits for the current technology used in computer systems. Multiprocessing (in a loose sense including distributed processing), therefore, becomes a natural and promising approach to explore.

Unfortunately, as in multi-person human teams, multiprocessing has problems. It may not work at all, for which the deadlock phenomenon [CES71], [IsM80] is a famous extreme example. While inappropriate algorithm or system design may be blamed for deadlock, there is also much inherent overhead in multiprocessing which degrades performance. Resource (memory, bus, function units, etc.) contention, precedence synchronization, critical-section lock-out, processor intercommunication cost, operating system overhead, etc. are all performance-degrading factors for multiprocessing.

This thesis mainly studies the memory interference problem (an important kind of resource contention in multiprocessor systems). Precedence synchronization and critical-section lock-out overhead is dealt with to a limited extent. The scope of the thesis is limited to tightly-coupled multiprocessor systems with shared main memory. Processor intercommunication in these systems is mostly accomplished through global variables stored in the shared memory; thus processor intercommunication cost is not a major concern in these systems.

The remainder of this chapter surveys previous work on the memory interference problem for both multiprocessor systems with time-division-multiplexed (TDM) busses and synchronous multiprocessor systems. Section 1.2 introduces Emer's model for multiprocessor systems with TDM busses and discusses various limitations of its applicability. Emer's model is generalized in Chapter 2 to remove these limitations. Section 1.3 discusses work by Skinner and Asher, Strecker, Ravi, Bhandarkar, Baskett and Smith, Rau, and Hoogendoorn for synchronous multiprocessor systems. A general model for the memory interference in such systems which allow arbitrary memory request rates, non-uniform memory references, and unequal processor priorities is presented in Chapter 3. An improved model for the case of uniform access and equal processor priority is also discussed. This model is further shown to be extendable to deal with multiprocessor systems where different memory service priorities are associated with different processor categories.

Chapter 4 presents several application examples which make use of the memory interference models derived in Chapters 2 and 3. However, the studies involved in these examples also have value in their own right. Section 4.2 proposes an algorithm for the estimation of the execution time of a program running in a multiprocessor system with and without memory interference. Section 4.3 discusses an investigation of the execution of matrix multiplication in a multiprocessor system with virtual memory.

Conclusions and suggestions for future work are summarized in Chapter 5.

Finally, an experiment for the effect of computation decomposition on the performance of executing Gaussian Elimination on a locally-designed and locally-built multiprocessor system, AMP-1 (see section 2.7), is presented in the Appendix.

1.2 Previous Work for Multiprocessor Systems with Time-Division-Multiplexed Busses and An Overview of Chapter 2

1.2.1 Background and Assumptions

Figure 1.1 shows an s-segment pipelined processor with a single fixed-cycle resource [Eme79]. At any time instant a distinct task is assumed to be active in each of the s distinct segments. A task, the schedulable entity for a pipeline, corresponds to one cycle of an instruction. Once a task enters the pipeline, it flows from segment to segment until it exits from the pipeline. Hence if we define each processor segment to take one segment time unit (STU) to perform its operation, each task will take s STUs for execution and s STUs thus compose one processor cycle time.

An instruction stream (i.e. a single program in execution) consists of an ordered sequence of instructions each of which is a sequence of pipeline tasks. At any given time the s tasks in the s distinct segments of the pipeline are assumed to come from s distinct instruction streams. The instruction streams could be totally independent of each other, working on totally independent jobs, or locally independent of each other while globally sharing some code and data. In the latter case, some interstream interactions may be required to achieve cooperation on a large multiprocessed job. The sharing and interaction mechanisms, if necessary, would be implemented explicitly by software.

Therefore, only one task from each instruction stream is active in

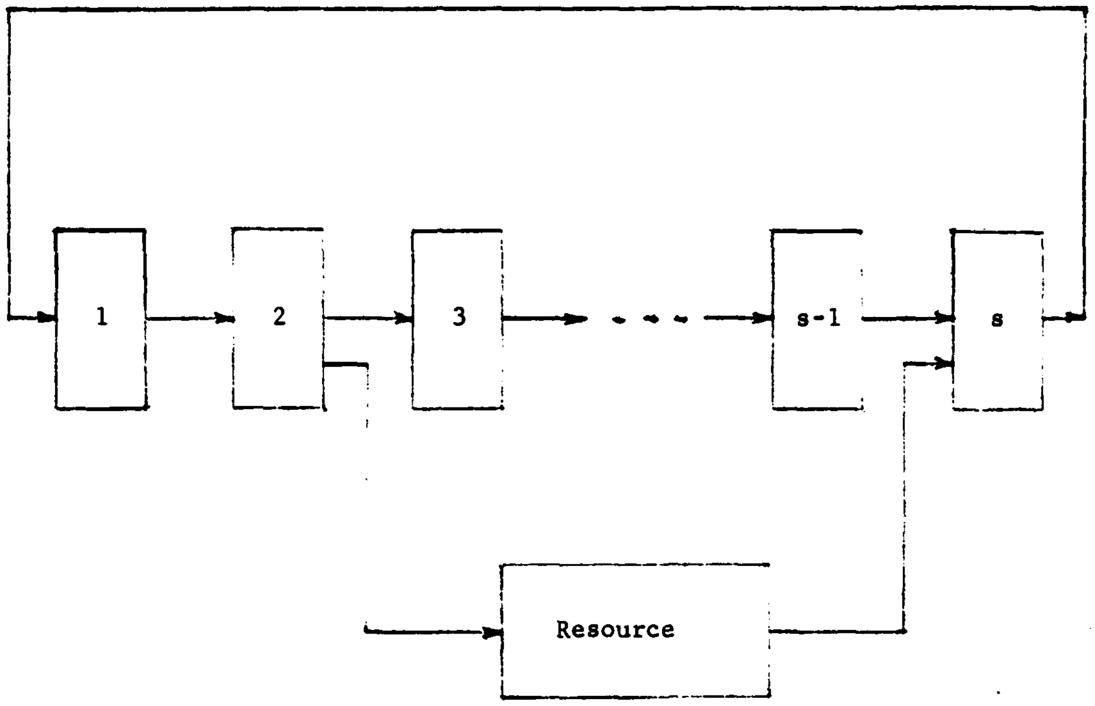


Figure 1.1 An s-segment Pipelined Processor with a Single Fixed-Cycle Resource

the pipeline at a particular time and all the tasks active simultaneously in the pipeline are independent of each other. This local independence alleviates the problem of data dependency which occurs frequently between tasks from the same instruction stream. It should be noted at this point, however, that a good decomposition of a complicated computation into multiprocessible streams cannot easily be achieved in general without deliberation. When a computation is not properly decomposed, the need for synchronization to solve global data dependency between instruction streams may become unnecessarily high. This shortens the period during which all instruction streams run smoothly and independently and thereby degrades performance.

Furthermore, the multiple-instruction-stream pipelined processor of Figure 1.1 is assumed to generate all requests to the shared resource from a particular segment. In other words, a task which requires the resource must make its resource request while it passes through that particular segment, say segment i. Any results produced by an accepted resource request will always be returned to the processor at segment i+j, where j-1 is the resource access time in terms of STUs, just as the task which originated the request arrives at that segment. No buffering is used in the system. In Figure 1.1, for example, a resource request could be generated at the end of segment 2 and the corresponding result, if any, will be returned by the resource to segment s if the request is accepted. The resource thus has an access time of (s-3) STUs.

Since for the purpose of performance evaluation, given the particular structure of the interconnection busses under consideration, there is no need to distinguish between the resource access time and the resource cycle time, we will hereinafter call this time period the resource cycle time and denote it by c. Any request submitted to the resource while it is busy serving a previously accepted request will be rejected. No service once in process by the resources under consideration can be pre-empted. Since the multiple-instruction-stream pipelined processor can generate at most one request per STU to the resource, a resource request will only be rejected when the resource is busy at the time the request is issued.

A rejected task simply takes a null pass through the pipeline and resubmits its request when it reaches the request-issuing segment in the next pass. This process must be repeated until the request is finally accepted.

In addition to the assumptions made above, the following important model assumption is imposed in order to obtain an analytic solution for the performance of such a system: all the requests generated from the pipelined processor, whether new or resubmitted, whether from different instruction streams or from the same instruction stream, are independent of each other and, if the resource is divided into several identical modules, are distributed uniformly among the modules.

When the resource models an interleaved memory with M memory modules, the assumption of independence between successive requests from the same instruction stream is, unfortunately, wrong in view of the sequentiality of the instruction and data request sequences [Rau77], especially those due to program constructs such as iteration and recursion. However, as pointed out in [Bha75] and [Rau79], the merging of instruction and data request sequences tends to reduce the effects of correlation between successive requests from the same instruction stream. The merging of various instruction streams in our pipelined processor further diminishes the effects of this correlation.

Tight iteration loops, which occur frequently in array computation algorithms, could nevertheless result in serious memory conflicts for a multiprogrammed computation executed by multiple processors. The independence assumption could be poor whenever this kind of conflict occurs. One should consider avoiding tight iteration loops in a multiprocessor program due to their adverse effect on performance. One straightforward approach is to unwind the tight loops by repetition so that the resultant strings are stretched as far as possible across the interleaved memory, as was done in [Leh66]. Conflict-free data array access is possible for SIMD-type array processors when data arrays are stored in a skewed fashion [BuK71], [Law75]. However, for MIMD-type multiprocessors in which processors do not work in a lock-step manner, algorithm modifications to distribute data accesses may be the only guideline available.

Furthermore, a resubmitted request is, of course, dependent on the preceding rejected request from the same instruction stream. In fact they are identical. However, since these two requests are separated by uncorrelated requests from other instruction streams and as long as the congestion that caused the original rejection has subsided, the reissued request would appear as if it were a new request. This assumption is particularly appropriate to requests generated by pipelined processors, since a rejected request is not reissued until one pass (s STUs) after it was rejected. The time between these requests should generally allow the original congestion to subside and permit the returning request to be viewed as a new request. In particular, these conditions should be satisfied in systems with good performance, since good performance implies few rejected requests. Because many rejections might tend to sustain congestion at the resource, this assumption could be less valid for systems with poor performance. Since we are mainly interested in systems with reasonably good performance, the assumption of independence within and among instruction streams will be followed throughout Chapter 2. Briggs[BrDa77] and Emer[Eme79] did show the robustness of this assumption by simulations.

Many researchers (e.g., [Str70], [Rav72], and [Bri77]), who studied the memory interference problem for the case in which processors´ memory request rates are equal to 1, ignored the rejected requests in their modeling work. This is appropriate for the study of the memory interference problem for the particular case they dealt with, as long as an instruction stream is considered as a sequence of independent requests. However, for the case in which processors´ memory request rates are less than 1, the independent request assumption with ignorance of rejected requests can lead to significant inaccuracies. If the rejected requests are totally ignored, then the instruction stream will not be perturbed and the memory request rate will be unchanged. The model, equation 3.22, presented in section 3.3 for the synchronous multiprocessor systems falls into this category. However, due to the resubmission of rejected requests, the memory request rate is indeed changed even if the instruction stream is still considered as a sequence of independent requests. One generally adjusts the memory request rate to compensate for this fact, as did Strecker[Str70] and Hoogendoorn[Hoo77] (described in the next section) and Emer[Eme79] (described below).

1.2.2 Emer's Model for a Multiple-Instruction-Stream Pipelined Processor with a Single Fixed-Cycle Shared Resource

Based on above assumptions Emer derived his analytic model for the performance of such a multiple-instruction-stream pipelined processor in terms of the following parameters: ψ, the probability that a task makes a resource request, α , the actual request rate seen by the resource (which is usually larger than ψ due to the contribution of reissued rejected requests), ρ , the number of passes the average task requires, and P_A, the probability of acceptance for a resource request.

The performance of the system is expressed by the parameter ρ , the number of passes the average task requires. This parameter is sometimes referred to as the interference factor for the system performance, because when the resource cycle time is equal to the processor cycle time it serves as a multiplicative factor for the program run time. Actually the reciprocal of ρ can be viewed as the probability that a pass or associated instruction stream is doing useful computation.

Since those tasks which do not request the use of the resource need only one pass through the pipeline while those tasks which do make resource requests may need more than one pass, the average number of passes a task may require can be evaluated as

$$\begin{aligned}\rho &= (1-\psi)\cdot 1+\psi[P_A\cdot 1+P_A(1-P_A)\cdot 2+P_A(1-P_A)^2\cdot 3+\cdots]\\ &= (1-\psi)+\psi P_A \frac{\partial}{\partial(1-P_A)}[(1-P_A)+(1-P_A)^2+(1-P_A)^3+\cdots]\\ &= (1-\psi)+\psi P_A \frac{\partial}{\partial(1-P_A)}\frac{1-P_A}{1-(1-P_A)}\\ &= 1-\psi+\psi\frac{1}{P_A}\ . \qquad (1.1)\end{aligned}$$

The derivation of the probability of acceptance for a resource request, P_A, can be argued intuitively. Since requests submitted to the resource are independent with a request rate of α , as seen by the resource, there will be an average of α(c-1) requests submitted to the resource during the c-1 STUs following an accepted request. All these requests will have to be rejected since the resource is busy serving that request. In other words, one request out of (1+ α(c-1)) requests will be accepted. Hence

$$P_A = \frac{1}{1+\alpha(c-1)}\ . \qquad (1.2)$$

Equation 1.2 can actually be derived rigorously. Emer [Eme79] refers to Briggs (p.100 of [Bri77]) who derived it in a much more general context. However, since Briggs' derivation is rather involved because of its generality and yet the request rate is restricted to be 1 instead of

ψ , a much simpler derivation based on the method of the imbedded Markov chain is given in the following: Looking at the system only at the very beginning of every STU (before any request to be submitted in that STU is presented), we define the system to be in state 0 if no request is in service by the resource and otherwise in state i if i STUs of service have been provided by the resource for the request in service. In Figure 1.2 is a discrete Markov chain model of the system. The resource of the system is or becomes idle in state 0, and has a probability α of seeing a request and accepting it. Once the service of a request is in progress, the resource remains busy for c STUs regardless what the processors are doing in the system. The steady state probability of state 0 becomes the probability that a submitted request is accepted.

If we use π_i to indicate the steady state probability of state i, $i=0,1,2,\ldots,c-1$, then we have

$$\pi_0 = \pi_{c-1} + (1-\alpha)\pi_0 \ . \tag{1.3}$$

$$\begin{aligned} \text{Also,} \quad & \pi_0 + \pi_1 + \pi_2 + \ldots + \pi_{c-1} \\ & = \pi_0 + (c-1)\pi_{c-1} \\ & = 1 \ , \end{aligned} \tag{1.4}$$

since $\pi_1 = \pi_2 = \ldots = \pi_{c-1}$.

Solving equations 1.3 and 1.4, we get

$$P_A = \pi_0 = \frac{1}{1+\alpha(c-1)} \ .$$

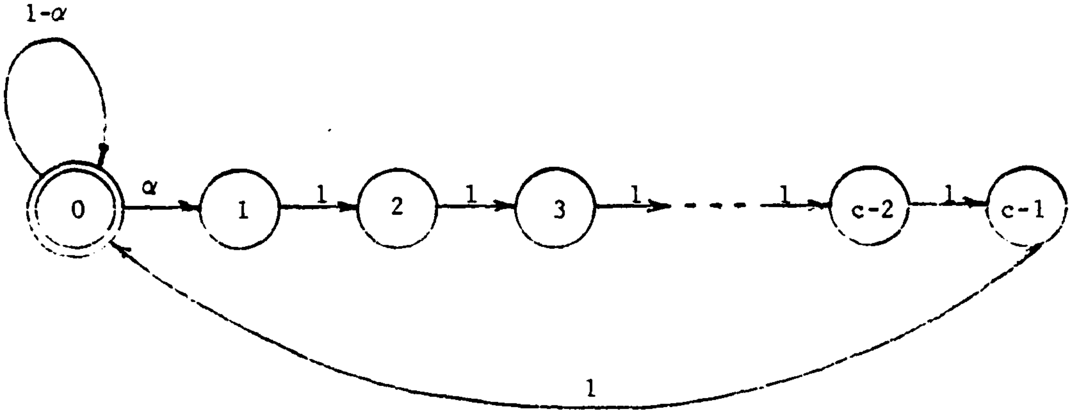


Figure 1.2 State Diagram for a Multiple-Instruction-Stream Pipelined Processor with a Single Shared Resource

Finally, the actual request rate, α , seen by the resource, can be determined by considering the total number of (identical) requests submitted to the resource by a task during the average number of passes it requires. More specifically,

$$\alpha = \frac{\psi[P_A \cdot 1+P_A(1-P_A)\cdot 2+P_A(1-P_A)^2\cdot 3 + \cdots]}{\rho}$$

$$= \frac{\psi \frac{1}{P_A}}{(1-\psi)+\psi \frac{1}{P_A}}$$

$$= \frac{1}{P_A(\frac{1}{\psi} - 1)+1} . \qquad (1.5)$$

Note that in terms of ψ and α a beautiful expression for the probability of acceptance can be obtained from equation 1.5. That is,

$$P_A = \frac{1/\alpha-1}{1/\psi-1} = \frac{\psi}{\alpha}\cdot\frac{1-\alpha}{1-\psi} . \qquad (1.6)$$

Using equations 1.1 and 1.6, one can express ρ in terms of ψ and α:

$$\rho = \frac{1-\psi}{1-\alpha} . \qquad (1.7)$$

Finally, using equations 1.2 and 1.5 together, one can also derive a closed-form expression for P_A as a function of ψ and c.

1.2.3 The Limitations of Emer´s Model and An Overview of Chapter 2

Emer´s model is of interest here because of the analogy between a multiple-instruction-stream pipelined processor system and a multiprocessor system with time-division-multiplexed busses. Emer´s model is simple and fairly accurately predicts the results of several simulations and some experimental data (section 2.8) taken from a real multiprocessor system, but it has several limitations.

First, it only models a single shared resource. This makes the model less useful for a system consisting of a number of different resources. A generalization which incorporates multiple distinct resources into the model is presented in section 2.2.

Secondly, the model, at least implicitly, restricts the resource cycle time to be less than or equal to one processor cycle time, i.e. the time a task requires to pass through the pipeline once (s STUs in Figure 1.1). This restriction is probably reasonable for control stores and main memory, but may be too restrictive for general function units and other types of resources which could be modeled. Section 2.3 relaxes this restriction on resource cycle times provided that the independent request assumption still holds.

Thirdly, in order to apply Emer´s model to a multiprocessor system with time-multiplexed busses, all the bus time slots must be assigned to active processors. This may not always be true. Some processors might be deactivated and some might be assigned by the operating system or the user to some other jobs totally unrelated to the activity we are considering. Hence such processors would not participate in the resource usage which we are investigating. These processors should be considered as non-existent as long as their existence does not interfere with the activity we are considering in any way. The time slots they occupy should be considered as vacant or inactive. However, if vacant or inactive time slots do exist, some parameters in the model will have to be modified since vacant time slots will never issue resource requests. A model extension dealing with "effective" parameter values and a vacant slot assignment problem, the "SCP problem", are presented in section 2.4.

Section 2.5 is mainly concerned with model modifications needed in order to include software resources as model resources. Some performance analysis concerning speedup and overhead according to the model of section 2.5 is done in section 2.6.

A matrix multiplication program was run on AMP-1, the multiprocessor machine to be described in section 2.7, and model-predicted values are compared to measured run time data in section 2.8. Also shown are the predicted values by two other methods - a deterministic model and a renewal-theory model - with and without modification by the memory interference factor. Values predicted with the modification match very well with the measured data.

1.3 Previous Work for Synchronous Computer Systems and An Overview of Chapter 3

In the scope of this thesis a synchronous computer system refers to a system in which all memory modules are cycled simultaneously and all processors are synchronized with the memory. The memory cycle time c is assumed to be 1. In other words, the operation of the system is assumed to be synchronized to the cyclic operation of the memory. All memory requests, if any, from all processors in the system are assumed to arrive at the beginning of a memory cycle. All previous work reviewed in the following is aimed at multiprocessor systems in this category.

1.3.1 Previous Work

Skinner and Asher[SKA69] proposed a discrete Markov chain model for multiprocessor systems with request rates equal to 1. Their analysis was presented for a small number of processors(≤ 2), and the model does contain tie-breaking probabilities in the case of memory usage conflict. However, for larger systems the complexity of the problem deterred the authors from further pursuit of an exact analytic solution.

Strecker[Str70], as also reported in [WuB72], developed a set of simple approximate models. For a system with N processors and M memory modules, the model corresponding to the case with request rates equal to 1 yields the now widely used formula for BW, the memory system bandwidth:

$$BW = M(1-(1-\frac{1}{M})^N) \quad . \tag{1.8}$$

Note that the memory bandwidth BW is the expected number of distinct memory modules requested, given N requests to M modules.

Formula 1.8 can easily be obtained by argument. Based on the assumption of N statistically identical processors and M identical memory modules with memory requests uniformly and independently distributed among them, (1-1/M) is the probability that a processor is not addressing a particular memory module. Then $(1-(1-1/M)^N)$ is the probability that at least one processor is accessing the particular memory module under consideration and thus is the probability for it to be busy. Collecting this probability for all M memory modules (multiplying the probability for one module by M since they are all identical), we get formula 1.8. The underlying assumption here is that the rejected requests are discarded to preserve the independence among requests. Another way of looking at this assumption, however, is that it is equivalent to removing the queued processors (those processors whose requests are rejected in the current memory cycle) from all the memory modules at the end of a memory cycle and reassigning them randomly among all the memory modules for possible service in the next cycle [Bha75] along with new requests from the nonqueued processors (those whose requests are accepted in this cycle).

Actually, it is this viewpoint that Strecker carried into the derivation of the bandwidth formula for the case in which memory request rates are allowed to be less than 1:

$$BW = M(1-(1-\frac{\alpha}{M})^N) \quad , \qquad (1.9)$$

where α is the adjusted request rate because of interference.

The fact that the queued processors are removed from all the memory modules and reassigned at the beginning of the next memory cycle independently and randomly among all the memory modules makes the bandwidth predicted by Strecker´s model overestimating, or optimistic. This point is further explained in section 3.3.

Ravi[Rav72] presented a similar model for the case in which request rates are equal to 1. He approached the memory interference problem by treating it as a combinatorial problem and presented the memory bandwidth in terms of the average number of distinct integers in a group of N integers chosen uniformly and independently from the integers 1 through M. Again, rejected requests were dropped in the derivation. Ravi´s somewhat complicated result has been shown to be exactly equal to formula 1.8[CKL77].

By employing a rather nice algorithm to evaluate the transition matrix of a Markov chain model, Bhandarkar[Bha75] was able to perform an exact analysis of the memory interference problem for the case in which request rates are equal to 1. Unfortunately, his algorithm is so time-consuming that the job of finding the memory bandwidth for a system with more than 16 processors and more than 16 memory modules becomes formidable.

Nevertheless, Bhandarkar did find from the exact analysis that the memory bandwidth is almost symmetric in N and M. Using this knowledge he modified Strecker´s memory bandwidth formula (formula 1.8) and made it a more accurate empirical estimate. That is,

$$BW = m[1-(1-\frac{1}{m})^n] \quad , \qquad (1.10)$$

where m = max(N,M) and n = min(N,M).

By viewing the memory modules as servers in a queueing system and assuming that the memory requests, the customer arrivals, are binomially distributed, Baskett and Smith[BaS76] obtained an expression that is asymptotically exact (as either M and/or N tend to infinity) for the system memory bandwidth. Although N and M were assumed to be very large in the derivation to make the state transition probability state-independent, the expression turned out to be fairly accurate even for small values of N and M. Their expression for the bandwidth is

$$BW = M+N-\frac{1}{2}-\sqrt{(M+N-\frac{1}{2})^2-2MN} \quad . \qquad (1.11)$$

The use of the binomial approximation was extended by Baskett and Smith[BaS76] to cover the case in which memory request rates may be less than 1. For this purpose they introduced the concept of "think time", which represents the period of time after a CPU receives the memory service previously requested and before it issues the next request. They were not able to solve for the mean queue length observed by a customer arriving at a memory module in this case, but they got around this by giving an educated and experienced guess. The approximate expression for memory bandwidth then becomes

$$BW = \frac{M}{2}(2+2L-\frac{1}{M}-\sqrt{(2+2L-\frac{1}{M})^2-8L}) \quad , \qquad (1.12)$$

$$\text{where } L = \frac{-(1+T-\frac{N-1}{M})+\sqrt{(1+T-\frac{N-1}{M})^2+4(\frac{N-1}{M})}}{2(\frac{N-1}{N})}\quad , \qquad (1.13)$$

and T is the mean of the think time distribution in units of memory service (cycle) time.

The accuracy of this result, as well as that of equation 1.9, is compared with simulation data for memory request rate in the range of (0,1] in section 3.3.

Recently, Rau[Rau79] used an approximation suggested by Baskett and Smith[BaS76] and obtained a very accurate closed-form expression of the memory bandwidth for the case in which request rates are equal to 1. The "decomposition approximation" simply states the following[Rau79]:

> All processors (totaling K) not queued at a given memory module are distributed among the other (M-1) modules with precisely the same distribution that would occur at equilibrium in a system consisting of K processors and (M-1) modules.

The intuitive justification for this assumption lies in the independence between the requests made by the various processors and the fact that each request has an equal and independent probability of being directed to any module[Rau79].

Rau´s bandwidth expression is

$$BW = \frac{\sum_{i=0}^{L-1} 2^i \binom{M-1}{i}\binom{N-1}{i}}{\sum_{i=0}^{L-1} \frac{2^i}{i+1}\binom{M-1}{i}\binom{N-1}{i}} \quad , \tag{1.14}$$

where L = min(N,M).

Again, this expression is symmetric in N and M, as is expression 1.11. As a matter of fact, expression 1.11 can be obtained from expression 1.14 via a further approximation[Rau79], namely the insensitivity of the memory bandwidth to the addition of one more module to the system when the value of M is very large.

Rau´s result is the best compared to models 1.8, 1.10, and 1.11 for the case in which request rates are equal to 1. Actually model 1.14 is about an order of magnitude more accurate (with respect to percentage error).

Until Hoogendoorn[Hoo77] presented his work, almost no researcher since Strecker and Ravi had attempted to attack the memory interference problem in a multiprocessor system by using a probabilistic approach. The probabilistic approach attacks the problem from observations of component behaviors, does not preserve the deterministic structure inherent in a closed queueing system, and hence does not usually produce very accurate results, as, for example, compared to Rau´s result obtained using a state-space approach. However, the probabilistic approach has its strong points. Because of the assumption of independence among memory requests made by the various processors, usually imposed in the probabilistic approach, the entire system is essentially decomposed into individual components. The complexity of the problem is, therefore, greatly reduced. Non-uniform request rates, request rates that are less than 1, and even processor priorities in accessing memory can thus be incorporated relatively easily into a probabilistic model without making the problem unmanageable.

Hoogendoorn[Hoo77] took advantage of this fact and presented a "general" model for memory interference in multiprocessors. In terms of a static access matrix, a dynamic access matrix, and the probabilities that particular processors are successfully accessing particular memory modules, his model consists of a set of nonlinear equations which has to be solved by iteration. Nevertheless, the model does allow arbitrary request rates for processors in the system. However, it is assumed that memory conflicts are resolved by an unbiased arbiter, so that when i processors attempt to access the same memory, each has a probability 1/i of success. Furthermore, for the case of uniform access the model reduces to model 1.9, which does not yield a very accurate estimate of the memory bandwidth (see section 3.3).

1.3.2 An Overview of Chapter 3

In section 3.1 the dependency class [CKL77] for the address streams produced by processors is specified. The desire for allowing arbitrary memory request rates and processor memory service priorities is also motivated in the same section.

Carrying over the same philosophy used in Chapter 2, we present in section 3.2 a general model based on the probabilistic approach for memory interference in synchronous multiprocessor systems. The model could be viewed as a cleaner version of Hoogendoorn's model with further generalization, although the two were derived independently.

In section 3.3 an attempt is made to improve the accuracy of the model by looking at the problem with the entire system in mind, as is done in the state-space approach, while still preserving the strong points of the probabilistic approach. The resulting model turns out to be fairly accurate over the entire range of request rates, except for systems with very few processors and memory modules. The predicted values of the memory bandwidth for various multiprocessor systems are compared with simulation data, together with predicted values derived from other models.

Finally, the model presented in section 3.3 can be used iteratively to deal with the memory interference problem in an environment where processor priorities in accessing memories are unequal. This application is presented in section 3.4.

CHAPTER 2

Memory Interference and Resource Contention in Computer Systems with Time-Division-Multiplexed Busses

2.1 Introduction

In this chapter, we consider the memory interference and resource contention in time-division-multiplexed (TDM) systems. The system has p identical processors. Time-multiplexed busses are used to interconnect these p processors with any system resources they share. For example, consider a common memory as a shared resource. Let it be attached to the system via an address bus, a read data bus, and a write data bus, shared by all the processors. A strict round-robin discipline is used to schedule the usage of these busses. All the processors in the system have the same processor cycle time, and a constant phase shift between clocks for successive processors is used. Appropriate phase shifts also exist between busses to account for the necessary processing delays. For example, Figure 2.1 shows the time slots assigned to processors for the usage of the shared address bus and the shared read data bus. The number indicated in each bus time slot is the processor number assigned to that slot. Note that the time slot assigned to a processor for the use of the read data bus is one memory access time after the time slot assigned to it for the use of address bus. If the memory module addressed by a

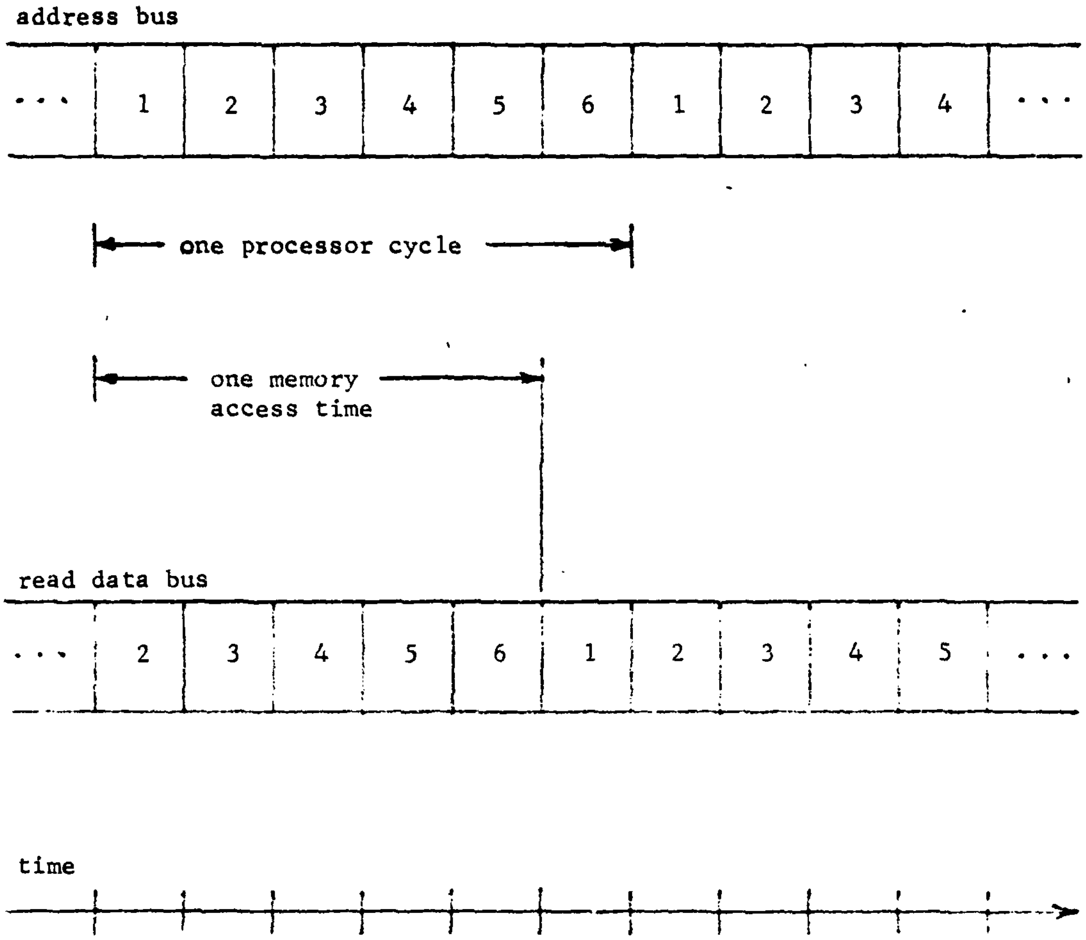


Figure 2.1 Illustration of the Time Slots Assigned to Processors for Bus Usage

processor is idle, the memory content read by the processor is put onto the read data bus at the next following read data bus time slot assigned to that processor. That read data bus slot will be left unused if the memory module addressed by this processor is busy. The write data bus operates similarly. (A bidirectional data bus may be used without conflict if the time between address and data bus time slots for a single request is the same for read and write transactions, Unfortunately this is usually not the case.)

Analytic tools will be refined in the following sections to evaluate the performance of such a system. The derivation will start from a model proposed by Emer [Eme79] for a multiple-instruction-stream pipelined processor with a single fixed-cycle shared resource (see section 1.2). Various generalizations are performed to enlarge the applicability of the model. These include generalizations for the number of shared resources (section 2.2), the relative magnitude (with respect to a processor cycle time) of the resource cycle times (section 2.3), the processor allocation in the round-robin bus window (section 2.4) and the type of shared resources (section 2.5). Model-based speed-up and overhead are analyzed in section 2.6. Finally, run time data from AMP-1, a multiprocessor computer system (see section 2.7), for a matrix multiplication program was measured to check with the predicted values from the model in section 2.8.

2.2 Generalization for A Multiple-Resource System

The concept of "resource" can be applied at many different levels in a computer system. The computer itself is a resource to outside users while, on the other hand, it is composed of various internal resources whose cooperation accomplishes the work demanded by the user. Whether a hardware or software entity should be considered as a resource depends on its participation in the activity under consideration at the level at which this activity is being modeled. When the attributes of an entity do not directly affect the activity that we are investigating, we will not consider that entity as a resource for this particular activity.

For example, a p-to-m crossbar switch in a system of p processors and m memory modules is not considered as a resource for the phenomenon of memory access conflict, since the switch itself cannot cause any conflict. The m memory modules should be considered as resources for the study of memory access conflict. However, if a full crossbar were replaced by a system with possible bus contention, then the bus entities and the memory modules should be considered as resources.

A more elaborate example involves the execution of a program on a multiprocessor system. Performance degradation could occur due to memory interference, critical section lock-out, etc. Therefore, for the performance evaluation of the multiprocessor system executing such a program, both the hardware memory modules and the critical sections of software should be considered as resources.

For above reasons the model derived in section 1.2.2 could be more useful if it would allow multiple resources in the system. This generalization is derived in this section.

Assume there are m resources in the system as in Figure 2.2, where resource i ($1 \le i \le m$) has a cycle time c_i no greater than the processor cycle time. Each task has a probability ψ_i of requesting resource i. Note that which particular segments are request-originating and result-receiving for particular resources is irrelevant to the model. We assume that the requests for distinct resources are independent and mutually exclusive, hence the sum of all ψ_i, $1 \le i \le m$, should be no greater than 1. The average number of passes a task would require now becomes

$$\begin{aligned} \rho &= (1 - \sum_{i=1}^{m} \psi_i) \cdot 1 + \sum_{i=1}^{m} \psi_i [P_{A_i} \cdot 1 + P_{A_i}(1 - P_{A_i}) \cdot 2 + \ldots.] \\ &= 1 - \sum_{i=1}^{m} \psi_i + \sum_{i=1}^{m} \psi_i \frac{1}{P_{A_i}} , \qquad (2.1) \end{aligned}$$

where the probability of acceptance for a request submitted to resource i is

$$P_{A_i} = \frac{1}{1 + \alpha_i (c_i - 1)} , \qquad 1 \le i \le m , \qquad . \qquad (2.2)$$

and the actual request rate seen by resource i is

$$\begin{aligned} \alpha_i &= \frac{\psi_i [P_{A_i} \cdot 1 + P_{A_i}(1 - P_{A_i}) \cdot 2 + \cdots]}{\rho} \\ &= \frac{\psi_i \frac{1}{P_{A_i}}}{1 - \sum_{j=1}^{m} \psi_j + \sum_{j=1}^{m} \psi_j \frac{1}{P_{A_j}}} , \qquad 1 \le i \le m . \qquad (2.3) \end{aligned}$$

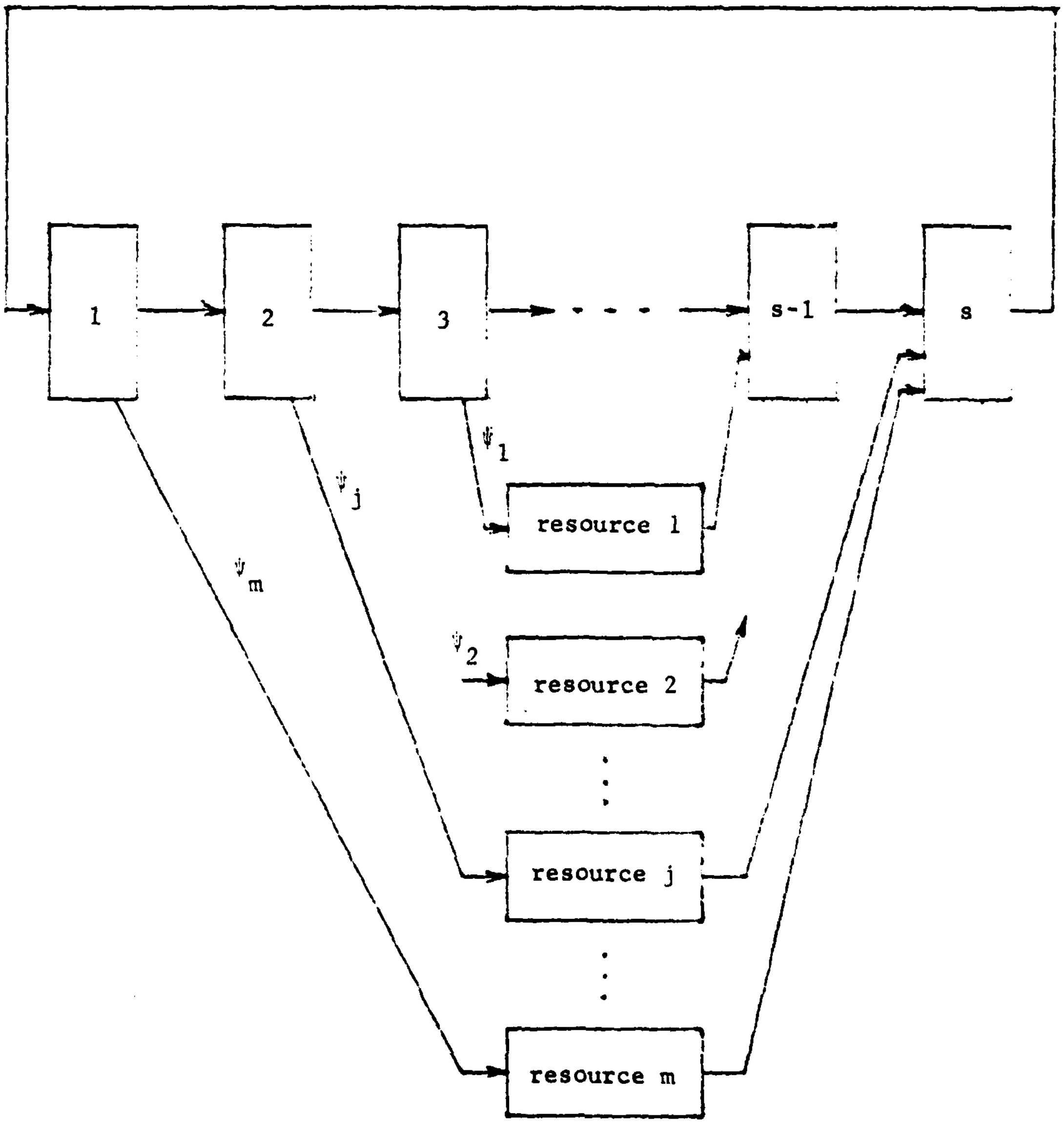


Figure 2.2 A Multiple-Resource Pipelined Processor

From here on, we call the programmed request rate to a resource, ψ , the static request rate to that resource. The actual request rate seen by the resource during run time, α , is called the dynamic request rate to that resource. Note that the dynamic request rate to resource i, α_i , is now not only a function of the static request rate to that resource, ψ_i, but also a function of the static request rates to the rest of the resources as well. The requests and usages of the resources in the system by the instruction streams are now interrelated.

To get some flavor of such a system, an example is given in Table 2.1. In systems 2 and 3, the pipeline is assumed to have more than 100 segments. Notice that system 3 is actually a system which contains both resources contained in system 1 and 2 and maintains the same static request rates to these resources from tasks. Due to program blockage caused by the rejection of a request to either resource, the dynamic request rate seen by the other resource is lowered. In this example, both α´s are actually lower in System 3. The probabilities of acceptance for requests to both resources are thus improved, but the overall performance, as reflected by the average number of passes a task requires, is worse, as should be expected. One may notice that the values of some parameters in this example have been exaggerated to show the coupling effect prominently, the performances of the systems are terribly poor, and hence the accuracy of the model due to its independence assumption is suspect in this case. It should be understood, though, that the example was constructed only for demonstration purposes.

Table 2.1

An Example Showing the Effect of the Existence of Multiple Resources

System 1

$\psi = 0.8, \quad c = 5$

$\alpha = 0.950515$

$P_A = 0.208244$

$\rho = 4.041647$

System 2

$\psi = 0.1, \quad c = 100$

$\alpha = 0.910100$

$P_A = 0.010977$

$\rho = 10.009957$

System 3

$\psi_1 = 0.8, \quad c_1 = 5$

$\psi_2 = 0.1, \quad c_2 = 100$

$\alpha_1 = 0.11623, \quad \alpha_2 = 0.87989$

$P_{A_1} = 0.68263, \quad P_{A_2} = 0.01135$

$\rho = 10.082821$

2.3 Generalization for the Resource Cycle Times

Up to this point all the resource cycle times have been restricted to be less than or equal to the processor cycle time. This restriction seriously reduces the generality of the model, because it excludes, for example, the possibility of modeling cache miss penalty (if only one miss can be processed at a time), complex function units, and critical sections. Each of these resources normally requires more than one processor cycle per access ($c > s$) and yet can only serve one request at a time. In this section we generalize the model to allow the resource cycle time to be longer than one processor cycle time.

Let n_i be a positive integer such that

$$(n_i-1)s < c_i \leq n_i s$$

where $i = 1, 2, \ldots, m$, m=the number of distinct resources in the system, $n_i \geq 1$, and s=the number of segments in the pipeline. Note that for resources modeled in previous sections, n_i=1.

Furthermore, we denote the delay between the issuance of a rejected request submitted to resource i and the issuance of the resubmitted request for the next trial as d_i processor cycles. In other words, d_i is not necessarily 1, as above. Allowing d_i to be greater than 1 is essential for modeling software resources, e.g. when a programmed request for accessing a critical section is rejected, it will usually take more than one processor cycle to loop through a few machine instructions and resubmit the request.

For this generalization, we have

$$\rho = (1 - \sum_{i=1}^{m} \psi_i)\cdot 1 + \sum_{i=1}^{m} \psi_i [P_{A_i} \cdot n_i + (1-P_{A_i})P_{A_i}(d_i+n_i)$$

$$+ (1-P_{A_i})^2 P_{A_i}(2d_i+n_i) + \cdots]$$

$$= 1 - \sum_{i=1}^{m} \psi_i + \sum_{i=1}^{m} \psi_i (n_i + d_i(\frac{1}{P_{A_i}} - 1)), \qquad (2.4)$$

$$\alpha_i = \frac{\psi_i [P_{A_i}\cdot 1 + (1-P_{A_i})P_{A_i}\cdot 2 + (1-P_{A_i})^2 P_{A_i}\cdot 3 + \cdots]}{\rho}$$

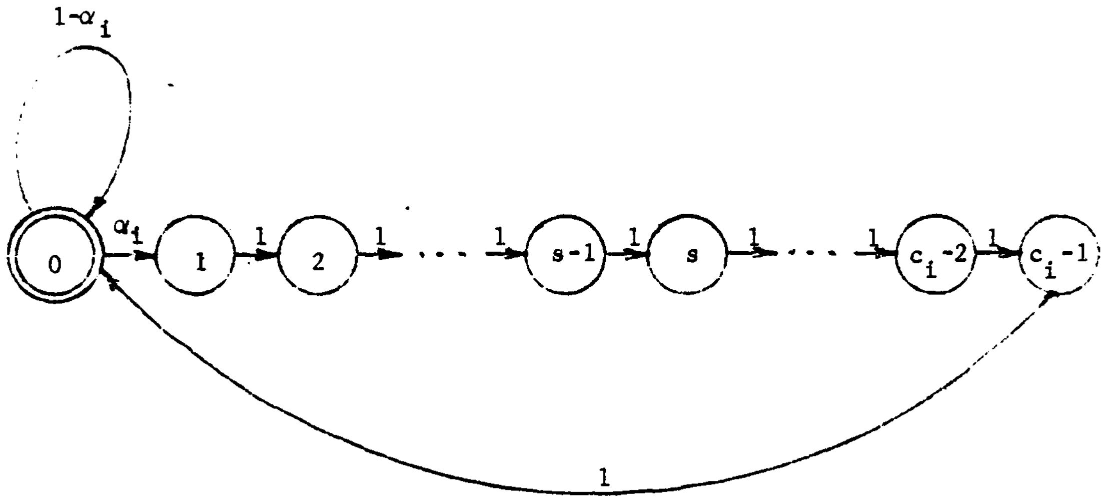
$$= \frac{\psi_i \frac{1}{P_{A_i}}}{\rho}$$

$$= \frac{\psi_i}{P_{A_i}[1 - \sum_{j=1}^{m} \psi_j + \sum_{j=1}^{m} \psi_j (n_j + d_j(\frac{1}{P_{A_j}} - 1))]}, \quad 1 \leq i \leq m \quad . \qquad (2.5)$$

Note that although the resource cycle time n_i and the retry delay d_i could be larger than 1, there will only be one request issued during each of those periods. Also, ρ could now assume a value larger than 1 even without resource contention.

Finally, the expression for the probability of acceptance for requests to resource i can be obtained by using a discrete Markov chain model (Figure 2.3) similar to that used in section 1.2.2. We thus have

$$P_{A_i} = \frac{1}{1 + \alpha_i (c_i - 1)} , \qquad 1 \leq i \leq m \quad . \qquad (2.6)$$



$(n_i-1)s < c_i \le n_i s$, where $n_i \ge 1$.

Figure 2.3 State Diagram for a Multiple-Cycle Resource

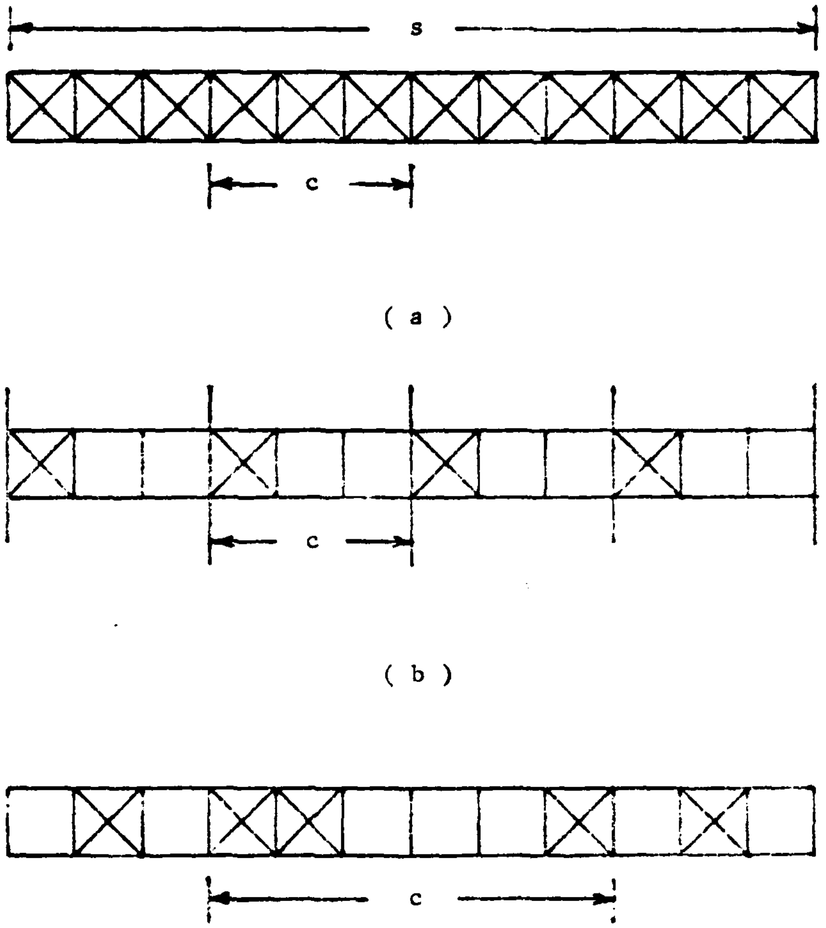
2.4 The SCP Problem

In this section we present a problem, which arises when the model discussed in previous sections has inactive instruction streams or unused time slots in a processor cycle.

The system we have been dealing with looks like the case in Figure 2.4 (a), where a cross signifies an occupied time slot by an active processor. The entire processor cycle is fully filled with active time slots. However, the resource cycle time c is used in the model thus far only to indicate that the next c-1 processors following an accepted request are locked out of the resource which accepts the request. When the processor cycle is fully filled, the number of processors locked out of the resource following an accepted request coincides exactly with the number of STUs the resource remains busy after it accepts the request.

Model modifications are required when there are vacant time slots corresponding to inactive instruction streams in the processor cycle. For example, in Figure 2.4 (b) only 4 out of the 12 processors are active and their assigned time slots are spaced evenly over the processor cycle as shown. The resource cycle time is still 3 STUs, as in (a). However, no processor will be blocked even if all processors always request the same resource. As far as resource access conflict is concerned, this system is equivalent to a 4-processor system with c=1 and 4 STUs per processor cycle, where the STU is 3 times as long as the STU in Figure 2.4 (b).

Therefore, in order to make Emer´s model applicable to this case, we



(a)

(b)

(c) p = 5 active processors

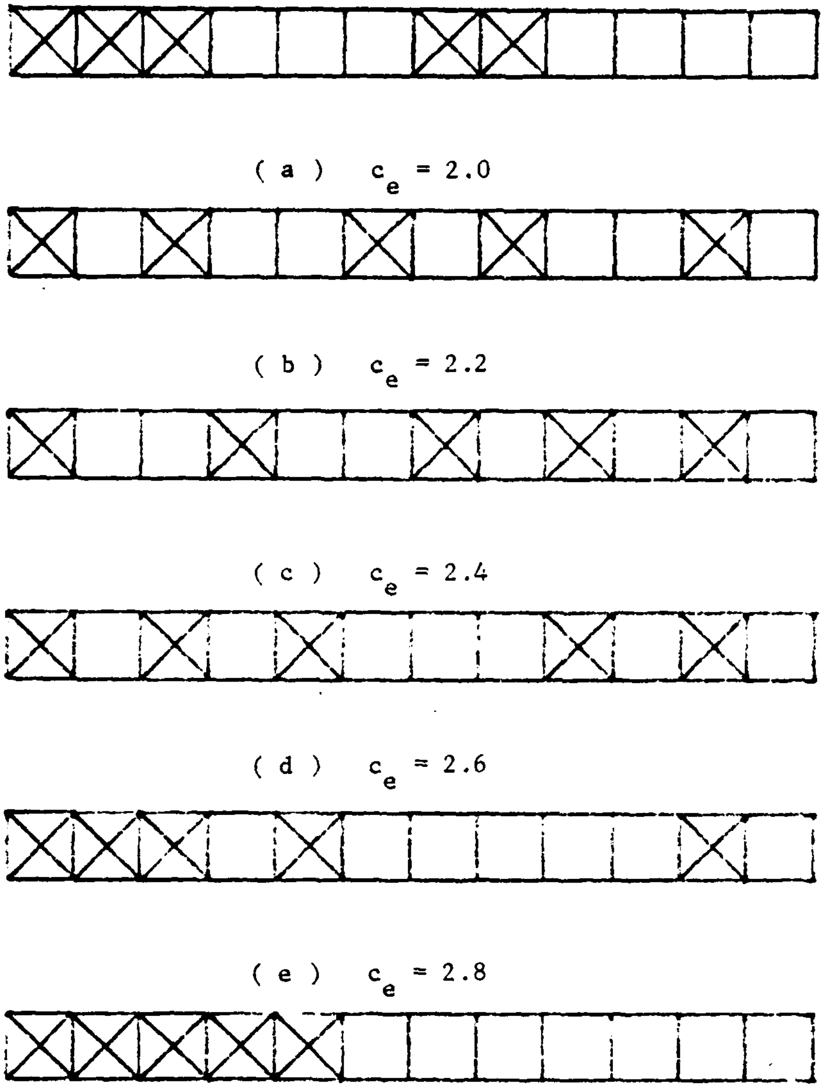
Figure 2.4 Illustrations for the SCP Problem

define an "effective resource cycle time", c_e, which assumes the role the resource cycle time plays in the model. This modification changes the relation between the c in Emer´s model and the physical time. In other words, if we use the effective resource cycle time c_e in lieu of the resource cycle time c in the model, the model will be able to cover this case.

More specifically, we define the effective resource cycle time, c_e, to be such that c_e-1 is the average number of active processors a processor could block. It should be noted that this simple averaging operation, instead of a rigorous Markov-chain-based argument, could cause some error. However, because of the close match between experimentally measured data and model-predicted values (see section 2.8), it is believed that the error is pretty small.

In the general case we have s time slots in one processor cycle, a physical resource cycle time of c STUs, and p active processors in the system. A brute-force approach for evaluating c_e-1 proceeds by summing up the total number of active processors which may be blocked by each processor and dividing the sum by p. For example, $c_e-1 = (2+2+1+2+2)/5 = 1.8$ in Figure 2.4 (c).

No simple formulas have been found which give the effective resource cycle time, c_e, directly. This is due to the difficulty of quantifying arbitrary allocations of the p active processors. Figure 2.5 gives an example to show the effect of different placements of the p active processors on the effective resource cycle time even with s, c, and p fixed. There do exist, however, special cases for which simple formulas



(a) $c_e = 2.0$

(b) $c_e = 2.2$

(c) $c_e = 2.4$

(d) $c_e = 2.6$

(e) $c_e = 2.8$

(f) $c_e = 3.0$

Figure 2.5 An Example of the Effect of the Processor Placement on the Effective Resource Cycle Time ($s = 12$, $c = 5$, $p = 5$)

exist for the effective resource cycle time.

For convenience, we will use the following definition:

Definition

A processor, say A, is said to be covered by another processor, say B, if the time slot assigned to processor A occurs less than one resource cycle time after the time slot assigned to processor B. In other words, processor A is covered by processor B if processor A is forbidden from using the resource whenever processor B is using it.

Case 1 $p = s$: $c_e = c$.

Since only one placement is feasible, the proof is straightforward.

Case 2 $c = s$: $c_e = p$

Proof: Since each processor covers all other processors, $c_e-1 = p-1$.

Case 3 $p = s-1$: $c_e = c-((c-1)/p)$

Proof: The placement pattern shown in Figure 2.6 (a) is the only feasible one (ignoring cyclically equivalent patterns). Then each processor will cover c-1 processors except the rightmost c-1 processors each of which covers one fewer because of the vacant slot at the right end. Therefore

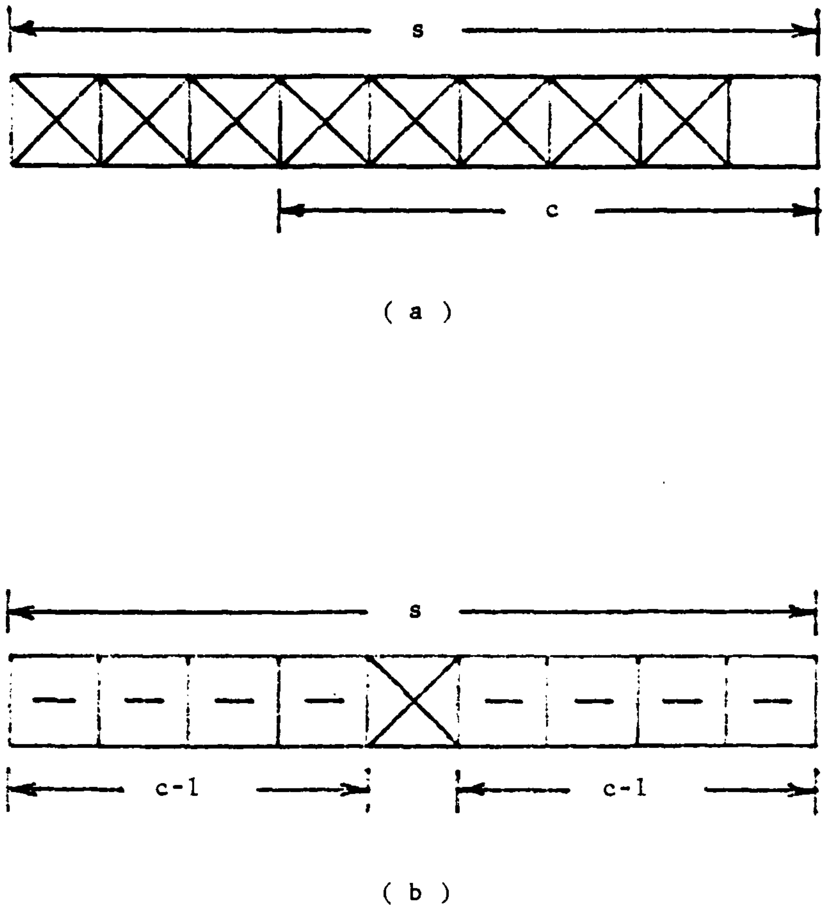


Figure 2.6 Illustrations for (a) $p = s-1$ (Case 3)
(b) $s = 2c-1$ (Case 4)

$$c_e - 1 = \frac{(p(c-1)-(c-1))}{p} = c - 1 - ((c-1)/p) \ . \qquad (2.7)$$

Case 4 $s = 2c-1$: $\quad c_e = (p+1)/2$

Proof: For each processor we look at its assigned time slot and the time period of length s with this time slot in the center (see Figure 2.6(b), where a time slot with a "-" sign means it is either occupied by an active processor or vacant.). Note that s is an odd integer in this case. All the active processors with assigned time slots located to the right of this processor are covered by this processor while this processor is covered by all the active processors with assigned time slots located to its left. Therefore, the contribution associated with this processor to the total sum of covered processors by all processors is p-1. This value actually includes the number of processors it covers and the number of processors by which it is covered. Therefore, multiplying p-1 by the total number of active processors, p, yields the product p(p-1) which is really twice the sum over all processors of the number of processors covered. Hence

$$c_e - 1 = (p(p-1)/2)/p = (p-1)/2.$$

It is interesting to note that in these cases c_e is independent of the allocation of the p active processors to time slots. The last case is of particular interest to us because the AMP-1, used in some experiments (see section 2.7), has s=9 and c=5 and thus falls into this category. Hence we can use the simple formula $c_e=(p+1)/2$ without worrying about the specific processor allocation.

Higher performance results from smaller effective resource cycle times since fewer processors are blocked due to resource access conflict. In general, c_e is a function of the particular allocation of active processors to time slots. Thus it would be nice if we could find an optimal allocation of active processors for any given s, c, and p. This information could be useful for the design of an operating system for the kind of machine we are considering. Unfortunately it turns out that finding an optimal allocation for a given s, c, and p is not a trivial problem and no general algorithm has been found.

We do have a formula, though, for the packed allocation (all the active processors are assigned consecutive time slots with the remaining time slots, if any, left vacant). The formula is given in the following:

(I) $p \leq c$

$$c_e-1 = (p-1)/2 + (1/2p)(p+c-s)\max(p+c-1-s,\ 0)$$

(II) $p > c$

$$c_e-1 = (c-1)(1-(c/2p)) + (1/2p)(p+c-s)\max(p+c-1-s,\ 0)$$

The derivation of this formula is discussed in two parts:

(1) $p + (c - 1) \leq s$

In this case processors in the packed time-slot chain can not cover processors preceeding them in time.

(i) $p \leq c$

$$c_e - 1 = \frac{(p-1)+(p-2)+\ldots+1+0}{p} = \frac{p(p-1)/2}{p} = \frac{p-1}{2}$$

(ii) $p > c$

$$c_e - 1 = \frac{(p-c)(c-1)+(c-1)+(c-2)+\ldots+1+0}{p}$$

$$= \frac{(p-c)(c-1)}{p} + \frac{c(c-1)/2}{p}$$

$$= \frac{c-1}{p}(p-\frac{c}{2}) = (c-1)(1-\frac{c}{2p})$$

(2) $p + (c - 1) > s$

In this case processors in the tail of the packed time-slot chain do cover processors at the head of the chain. Note that $s-(c-1)$ is the number of processors at the head of the chain which do not cover any processor preceeding them; $p-(s-(c-1))$ is the number of processors in the tail of the chain which do cover some processors preceding them.

(i) $p \leq c$

$$c_e - 1 = \frac{(p-1)+(p-2)+\ldots+[p-(s-(c-1))]+[p-(s-(c-1))]^2}{p}$$

$$= \frac{1}{p}[\frac{p(p-1)}{2} - \frac{[p-(s-(c-1))][p-(s-(c-1))-1]}{2}$$

$$+ [p-(s-(c-1))]^2]$$

$$= \frac{p-1}{2} + [p-(s-(c-1))]\frac{1}{p}\{[p-(s-(c-1))] - \frac{p-(s-(c-1))-1}{2}\}$$

$$= \frac{p-1}{2} + \frac{1}{2p}(p+c-s)(p+c-1-s)$$

(ii) $p > c$

$$c_e - 1 = \frac{(p-c)(c-1)+(c-1)+(c-2)+\ldots+[p-(s-(c-1))]+[p-(s-(c-1))]^2}{p}$$

$$= \frac{(p-c)(c-1)}{p} + \frac{c(c-1)/2}{p} + \frac{1}{2p}(p+c-s)(p+c-1-s)$$

$$= (c-1)(1-\frac{c}{2p}) + \frac{1}{2p}(p+c-s)(p+c-1-s)$$

The formulas for cases (1) and (2) can be combined together and the result is as given previously.

Before we present the optimality theorem for the packed allocation, we need the following lemma:

Lemma

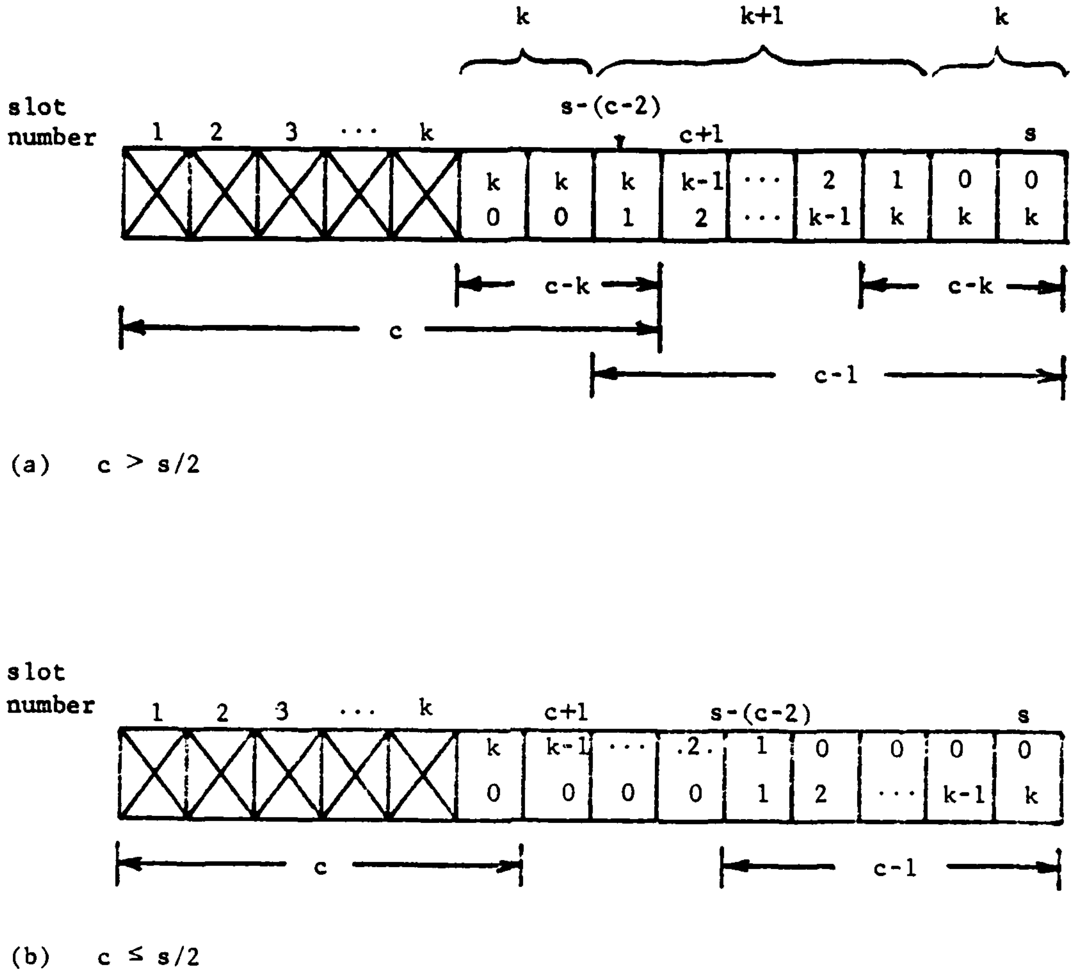
Given a system of k $(0<k<s)$ active processors placed in a packed allocation. If one extra processor is to be added to the system, then the $(k+1)$-processor packed allocation yields, among all possible positions for processor $k+1$, the minimum effective resource cycle time for the case $c > s/2$, and yields the maximum effective resource cycle time for the case $c \leq s/2$.

Proof:

If one extra processor is to be added to a k-processor system, the sum of the total number of active processors which would be blocked by other processors will be increased by two kinds of contributions: the sum of those processors which would block this new processor and the sum of those processors which this new processor would block. In Figure 2.7 the former value is shown above in the potential time slot for this processor and the latter is shown below. Figure 2.7 is valid for the $k \leq c-1$. The case $k \geq c$ will be discussed later.

Note that from left to right (increasing time) the values in the top sequence remain at k first, start to decrease at time slot c+1 with a slope of -1 (decrease by 1 per time slot), and remain at 0 when the values decrease to 0. On the contrary, the values in the bottom sequence remain at 0 first, start to increase at time slot s-(c-2) with a slope of +1 (increase by 1 per time slot), and remain at k when the values increase to k. Thus the sum of the two sequences is one which assumes the value k at both ends with values in the middle greater than, equal to, or less than k depending on the relative magnitudes of c+1 and s-(c-2).

When $s-(c-2) < c+1$, the bottom sequence starts to increase earlier than the top sequence starts to decrease and it can be shown that the sum of the two sequences assumes values greater than k in the middle (see Figure 2.7(a)). When $s-(c-2) = c+1$, i.e. the two sequences change slope simultaneously, the sum of the two sequences assumes the value k throughout the c-k time slots. This case actually corresponds to Case 4 above, which requires $s = 2c-1$. In either case, a packed (k+1)-processor



(a) $c > s/2$

(b) $c \leq s/2$

Figure 2.7 Contributions Due to an Extra Processor Added to a k-processor Packed Allocation when $k = c-1$

allocation yields the minimum effective resource cycle time (with the minimum extra contribution k from processor k+1). Note that $s-(c-2) \leq c+1$ corresponds to $c \geq (s+1)/2$, which is exactly the condition $c > s/2$.

On the other hand, when $s-(c-2) > c+1$, i.e. the top sequence starts to decrease earlier than the bottom sequence starts to increase, it can be shown that the sum of the two sequences assumes values less than k in the middle (see Figure 2.7(b)). The condition of this case is exactly the condition $c \leq s/2$ and a packed (k+1)-processor allocation in this case yields the maximum effective resource cycle time (with the maximum extra contribution k from processor k+1).

Finally it can be shown that similar argument with trivial modification holds for the case $k \geq c$. The proof is thus omitted.

Q.E.D.

Finally, we have the following theorem:

Theorem

Given $p \leq c$, the packed allocation of active processors yields the minimum effective resource cycle time for the case $c > s/2$, and yields the maximum effective resource cycle time for the case $c \leq s/2$.

Proof:

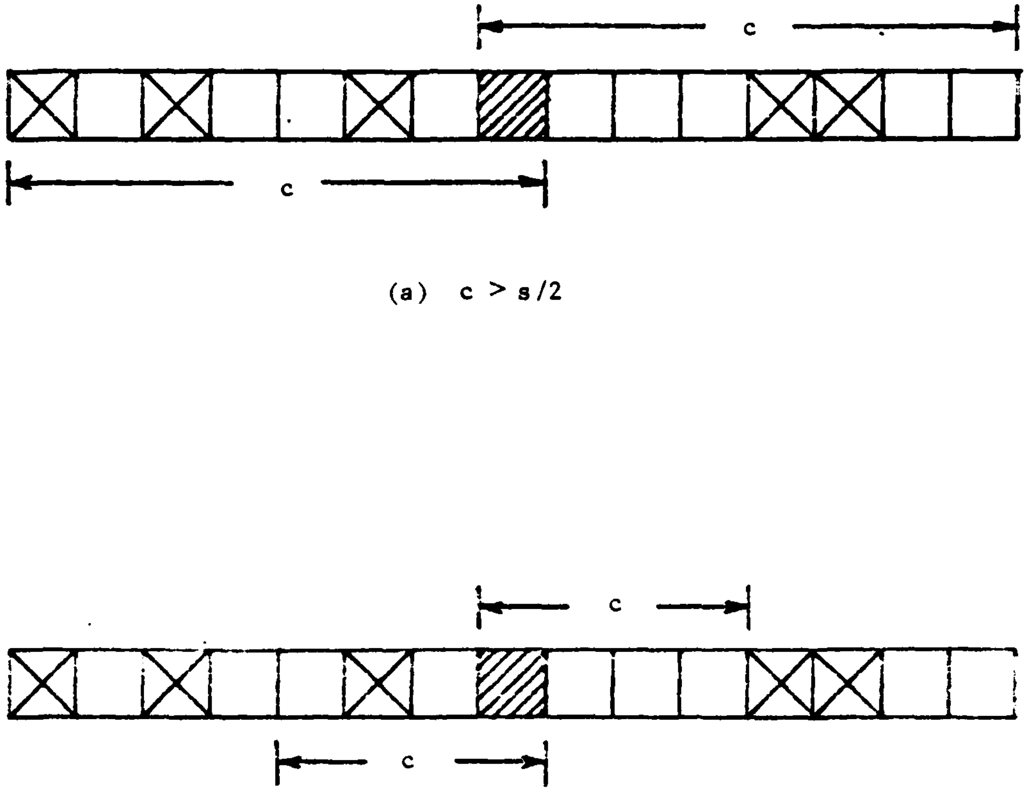
The case of 1 processor is uninteresting. Thus, suppose we have 2 processors in the system. If $c > s/2$, i.e. $2c > s$, conflict between processors can not be avoided and the packed allocation yields the minimum effective resource cycle time. On the other hand, if $c \leq s/2$, i.e. $2c \leq s$, conflict can be completely avoided. The packed allocation in this case yields the maximum effective resource cycle time.

Assume the theorem is valid for k processors, where $2 \leq k \leq c-1$.

Now consider adding one extra processor to a k-processor system. Let us concentrate on the contribution due to this new processor to the sum of the total number of active processors which would be blocked by other processors. If the k processors are placed in a packed allocation, it is known from the lemma above that putting the new processor in such a position to form a (k+1)-processor packed allocation will, among all possible positions, achieve minimum contribution for the case $c > s/2$ and the maximum contribution for the case $c \leq s/2$.

If the k processors are not placed in a packed allocation, we will show in the following for the case $k \leq c-1$ that the new contribution due to processor k+1 will be at least the amount it contributes if it were added to a k-processor packed allocation for the case $c > s/2$ and at most the amount it contributes if added to a k-processor packed allocation for the case $c \leq s/2$. This fact together with the inductive assumption yields the theorem for k+1 processors, provided $k \leq c-1$.

When $c > s/2$, we have $2c-1 \geq s$. Therefore, no matter where processor k+1 is placed into a k-processor allocation, the total number of processors which it covers and which cover it will be at least k (see Figure 2.8, where the shaded time slot indicates a potential position for



(a) c > s/2

(b) c ≤ s/2

Figure 2.8 An Extra Processor Added to a k-processor Unpacked Allocation

processor k+1). On the other hand, when $c \leq s/2$, we have $2c-1 \leq s-1$. Therefore, no matter where processor k+1 is placed into a k-processor allocation, the total number of processors which it covers and which cover it will be at most k (see Figure 2.8). Remember from the proof of the lemma that k is the minimum and the maximum extra contribution for the case $c > s/2$ and $c \leq s/2$, respectively, for the case $k \leq c-1$ when processor k+1 is added to a k-processor packed allocation. This extra contribution k is achieved by the (k+1)-processor packed allocation in both cases.

Therefore, while $k \leq c-1$, when an extra processor is added to a k-processor allocation, packed or unpacked, the minimum extra contribution in the case $c > s/2$ and the maximum extra contribution in the case $c \leq s/2$ are both k. Since by assumption the packed allocation of k active processors yields, among all possible k-processor allocations, the minimum effective resource cycle time for the case $c > s/2$ and the maximum effective resource cycle time for the case $c \leq s/2$, the theorem is apparently valid for k+1 processors.

By mathematical induction the theorem is valid for any number of processors smaller than or equal to c.

Q.E.D.

2.5 Generalization for Software Resources

In a multiprocessor program the execution of certain segments of the program code must be controlled in order to protect data integrity and/or to provide synchronization information. Very often protection is provided by allowing only one processor at a time to execute these segments and no preemption is allowed. Then such segments are usually referred to as mutual-exclusion critical sections or, loosely, just critical sections. Since processors have to compete to gain access to these critical sections, they are essentially shared resources of the system.

However, while a processor is executing program code, it also accesses the memory - another shared resource in the system. In the model discussed in section 2.3 a task needs service from at most one resource in the system. When a resource i request from a task is accepted, no request to any resource from the same task will be submitted during the following c_i-1 time slots. For a rejected resource i request, on the other hand, the task under consideration must submit no resource request during the d_i-1 processor cycles following a previously rejected request. The last processor cycle in the d_i-cycle period is used to resubmit the resource i request. Therefore, if we try to model a mutual-exclusion critical section as a resource of a multiprocessor system, these constraints will have to be relaxed. However, when a shared resource is indeed the kind of resource modeled in section 2.5, as is often the case for a function unit, the previous model should be left intact. In the rest of this chapter, we will call shared software resources, like critical section code, Type-0 resources. The kind of shared resource modeled previously is called a Type-1 resource hereafter. The value of parameter δ_i will be used to indicate the type of resource i.

Without loss of generality, we will let resource 0 be the shared common memory in this section and we assume there are η identical memory modules in the memory. The requests to memory modules are assumed to be distributed randomly and independently. This assumption may be closely approximated in practice by fully interleaved addressing as discussed in section 1.2.1.

Finally, let's assume there are m non-memory shared resources and p active processors in the system.

Note that in the particular cycle in which a processor requests a non-memory resource, it is assumed not to be requesting the memory simultaneously. In other words, the events to which ψ_i probabilities are assigned are mutually exclusive. Hence if a non-memory resource i is a Type-0 resource its resource cycle time n_i will be expanded on the average due to memory interference to be

$$1+(n_i-1)(1-\psi_0+\psi_0\frac{1}{P_{A_0}})$$

since only the last n_i-1 processor cycles could possibly expand due to memory interference. If the non-memory resource i is a Type-1 resource, however, the resource cycle time n_i will not be affected by memory interference. By means of the type parameter δ_i , the two resource cycle times can be combined and the resulting resource cycle time for resource i becomes

$$1+(n_i-1)(\delta_i+(1-\delta_i)(1-\psi_0+\psi_0\frac{1}{P_{A_0}}))\ .$$

Furthermore, it should be noted that since fully interleaved addressing is assumed the memory reference behavior is assumed to be so homogeneous that the memory request rates within and outside Type-0 resource cycles are the same.

The average number of passes a task requires, ρ , now becomes

$$\rho = (1-\psi_0-\sum_{i=1}^{m}\psi_i)\cdot 1+\psi_0[P_{A_0}\cdot 1+(1-P_{A_0})P_{A_0}\cdot 2+\cdots\ ]$$

$$+\sum_{i=1}^{m}\psi_i\{P_{A_i}[1+(n_i-1)(\delta_i+(1-\delta_i)(1-\psi_0+\psi_0\frac{1}{P_{A_0}}))]$$

$$+(1-P_{A_i})P_{A_i}[2+((n_i-1)+(d_i-1))(\delta_i+(1-\delta_i)(1-\psi_0+\psi_0\frac{1}{P_{A_0}}))]$$

$$+\cdots\}$$

$$= 1-\psi_0-\sum_{i=1}^{m}\psi_i+\psi_0\frac{1}{P_{A_0}}$$

$$+\sum_{i=1}^{m}\psi_i\{\frac{1}{P_{A_i}}+[(n_i-1)+(d_i-1)(\frac{1}{P_{A_i}}-1)](\delta_i+(1-\delta_i)(1-\psi_0+\psi_0\frac{1}{P_{A_0}}))\} \qquad (2.8)$$

The request rate seen by the memory as a whole entity becomes

$$\alpha_0 = \frac{\psi_0 \frac{1}{P_{A_0}} + \sum_{i=1}^{m} \psi_i [P_{A_i}(n_i-1)\psi_0(1-\delta_i)\frac{1}{P_{A_0}} + (1-P_{A_i})P_{A_i}[(n_i-1)+(d_i-1)]\psi_0(1-\delta_i)\frac{1}{P_{A_0}} + \cdot\cdot]}{\rho}$$

$$= \frac{\psi_0\{1+\sum_{i=1}^{m} \psi_i[(n_i-1)+(d_i-1)(\frac{1}{P_{A_i}} -1)](1-\delta_i)\}}{P_{A_0}\rho} \tag{2.9}$$

The request rate seen by a non-memory resource is

$$\alpha_i = \frac{\psi_i[P_{A_i}\cdot 1 + (1-P_{A_i})P_{A_i}\cdot 2 + \cdots]}{\rho}$$

$$= \frac{\psi_i}{P_{A_i}\rho} , \qquad i=1,2, \ldots,m . \tag{2.10}$$

The requests for memory are actually distributed among all η modules. Hence for a particular module the request rate for it is one ηth of the α_0 we obtained above.

The probability of acceptance for a memory request now becomes

$$P_{A_0} = \frac{1}{1+(\alpha_0/\eta)(c_e-1)} \tag{2.11}$$

while the probability of acceptance for a non-memory resource request is

$$P_{A_i} = \frac{1}{1+\alpha_i\{[1+(n_i-1)(\delta_i+(1-\delta_i)(1-\psi_0+\psi_0\frac{1}{P_{A_0}}))]p-1\}} , \tag{2.12}$$

$$i=1,2, \ldots,m .$$

2.6 Speedup and Overhead

In this section, we extract some performance information out of the model discussed in the previous section.

An array computation program intended to run in a multiprocessor environment often consists of sections each with the structure shown in Figure 2.9[Che71]. There, a small segment of code at the beginning of the section is usually devoted to initializing the process. This initialization segment is then followed by a good number of virtually independent jobs of equal or unequal sizes. Some jobs in this job pool may have precedence dependency among them, but the degradation effect of the dependency could be implicitly removed by properly scheduling the processors.

Every processor participating in executing this section of program code must either execute the initialization segment or wait in a busy-wait loop while that segment is being executed. Thus the initialization segment is not sharable. However, if there is more than one processor in the system, they can share the load of executing the following parallel jobs. Let R_s be the total number of programmed processor cycles in the initialization segment. Let R_0 be the total number of programmed processor cycles in all the other jobs. Since the number of jobs is usually large compared to the number of processors, we will optimistically assume that R_0 can be evenly shared.

Let ρ_1 and ρ_p be the average number of cycles a task requires in a single processor and p-processor environment, respectively. Then the run

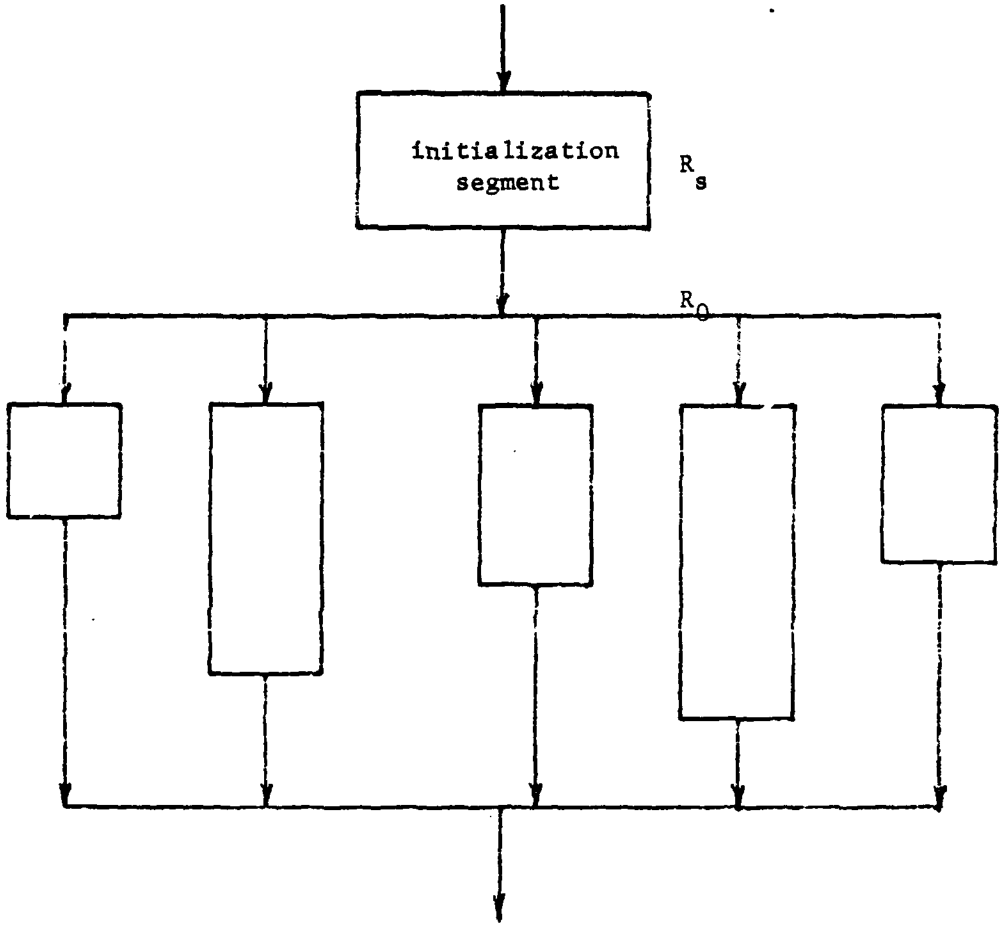


Figure 2.9 Structure of a Program Section in a Multiprogrammed Computation

time in processor cycles for the execution of the program section using a single processor is

$$R_1 = (R_s + R_0)\rho_1 \quad . \tag{2.13}$$

On the other hand, the run time in processor cycles for the execution of the program section using p processors is

$$R_p = (R_s + \frac{R_0}{p})\rho_p \quad . \tag{2.14}$$

Therefore, the speed-up, S, for using p processors instead of one can be calculated as

$$S = \frac{R_1}{R_p} = \frac{(R_s + R_0)\rho_1}{(R_s + \frac{R_0}{p})\rho_p} = p\,\frac{(R_0 + R_s)\rho_1}{(R_0 + pR_s)\rho_p} = \frac{p}{\lambda} \tag{2.15}$$

$$\text{where } \lambda = \frac{(R_0 + pR_s)\rho_p}{(R_0 + R_s)\rho_1} = \frac{(1 + p\frac{R_s}{R_0})\rho_p}{(1 + \frac{R_s}{R_0})\rho_1} \tag{2.15}$$

If the initialization part takes a negligible amount of time compared to the sharable part of the program, i.e. $PR_s << R_0$, then the "penalty factor" [Kun76], λ , can be expressed approximately as

$$\lambda \approx \frac{\rho_p}{\rho_1} \quad . \tag{2.16}$$

It may now be instructive to examine various overhead ingredients in the expression for the average number of passes required by a task, ρ , according to the model derived in section 2.5.

The general expression for ρ is (equation 2.8)

$$\rho = \rho(P_{A_0}, P_{A_i}, \; i=1,2, \; \ldots, m)$$

$$= 1 - \psi_0 - \sum_{i=1}^{m} \psi_i + \psi_0 \frac{1}{P_{A_0}}$$

$$+ \sum_{i=1}^{m} \psi_i \{ \frac{1}{P_{A_i}} + [(n_i - 1) + (d_i - 1)(\frac{1}{P_{A_i}} - 1)][\delta_i + (1-\delta_i)(1-\psi_0 + \psi_0 \frac{1}{P_{A_0}})]\} \tag{2.17}$$

Even with no resource contention at all, we have this no-overhead value for ρ :

$$\rho_0 = \rho(P_{A_0} = P_{A_i} = 1, \; i=1,2, \; \ldots, m)$$

$$= 1 - \sum_{i=1}^{m} \psi_i + \sum_{i=1}^{m} \psi_i n_i \tag{2.18}$$

As before, resource 0 is used to refer to memory. Hence the overhead purely due to memory contention is

$$\xi_0 = \rho(P_{A_0} < 1,\ P_{A_i} = 1,\ i=1,2,\ldots,m) - \rho_0$$

$$= \psi_0\left(\frac{1}{P_{A_0}} - 1\right)\left[1 + \sum_{i=1}^{m} \psi_i (n_i - 1)(1 - \delta_i)\right] \quad . \qquad (2.19)$$

This additive overhead is directly proportional to ψ_0, the probability that a task makes a memory request. The second term in the square bracket represents the overhead factor due to a processor executing code during the execution of a software resource (e.g. a critical section).

The overhead due directly to non-memory resource contention can be expressed as

$$\xi_j = \rho(P_{A_0} = 1,\ P_{A_j} < 1,\ P_{A_i} = 1,\ i=1,2,\ldots,m,\ i \neq j) - \rho_0$$

$$= \psi_j d_j\left(\frac{1}{P_{A_j}} - 1\right) , \qquad j=1,2,\ldots,m \quad . \qquad (2.20)$$

Again, this overhead is proportional to the request rate, ψ_j. Note that it is also proportional to the retry delay d_j.

It is interesting to note the existence of some coupled overhead between memory contention and non-memory software resource contention. This coupling is due to memory contention interfering with software resource retrials.

The coupled overhead is

$$\begin{aligned}\xi_{c0} &= \rho - \xi_0 - \sum_{i=1}^{m} \xi_i - \rho_0 \\ &= \psi_0\left(\frac{1}{P_{A_0}} - 1\right) \sum_{i=1}^{m} \psi_i (d_i - 1)\left(\frac{1}{P_{A_i}} - 1\right)(1 - \delta_i)\end{aligned} \qquad (2.21)$$

Finally, the interference factor ρ can thus be expressed in terms of the basic no-overhead value and various overheads:

$$\rho = \rho_0 + \xi_0 + \sum_{i=1}^{m} \xi_i + \xi_{c0} \quad . \qquad (2.22)$$

2.7 An Experimental Tool - the AMP-1 Machine

In this section we briefly describe a locally-designed and locally-implemented multiprocessor computer system, the AMP-1 machine[Dav80], [Hor78], [Kra77], which is used in the next section and the Appendix of the thesis to study the inherent resource contention and job multitasking for parallel execution in a multiprocessor system.

2.7.1 System Organization

A block diagram of the AMP-1 system [Hor78], [Kra77] appears in Figure 2.10. The following description is mainly taken from [Dav80].

The system employs eight Motorola 6800 microprocessors which access memory over a shared bus system using a strict round-robin bus window access discipline. The processor controller (see Figure 2.10) contains the master clock which generates control signals for the processors and their associated external registers and drivers. The shared bus system consists of an address bus, a read data bus and a write data bus. Shift registers in the processor controller provide the necessary control signal skew between successive processors, so the processors take turns in using these busses in a strict round-robin fashion. For a particular processor, appropriate time delay exists between its address bus window and its read or write data bus window so that the memory has enough time to process its request, if accepted.

The memory is organized as 64 modules of 1K bytes each. When operating at full speed, the memory modules have a cycle time of 5, i.e. when a processor accesses a memory module the next 4 processors in sequence are forbidden from accessing that memory module. The busy checker determines whether a memory access request is attempting to access a busy module and if so, disables the clock for that processor for one complete cycle. The design of the Motorola 6800 processor and the clock disable logic permit a rejected memory access request to be resubmitted automatically on the next cycle. The busy checker permits an extension of the memory cycle time beyond 5 clock times to 6, 7, or 8

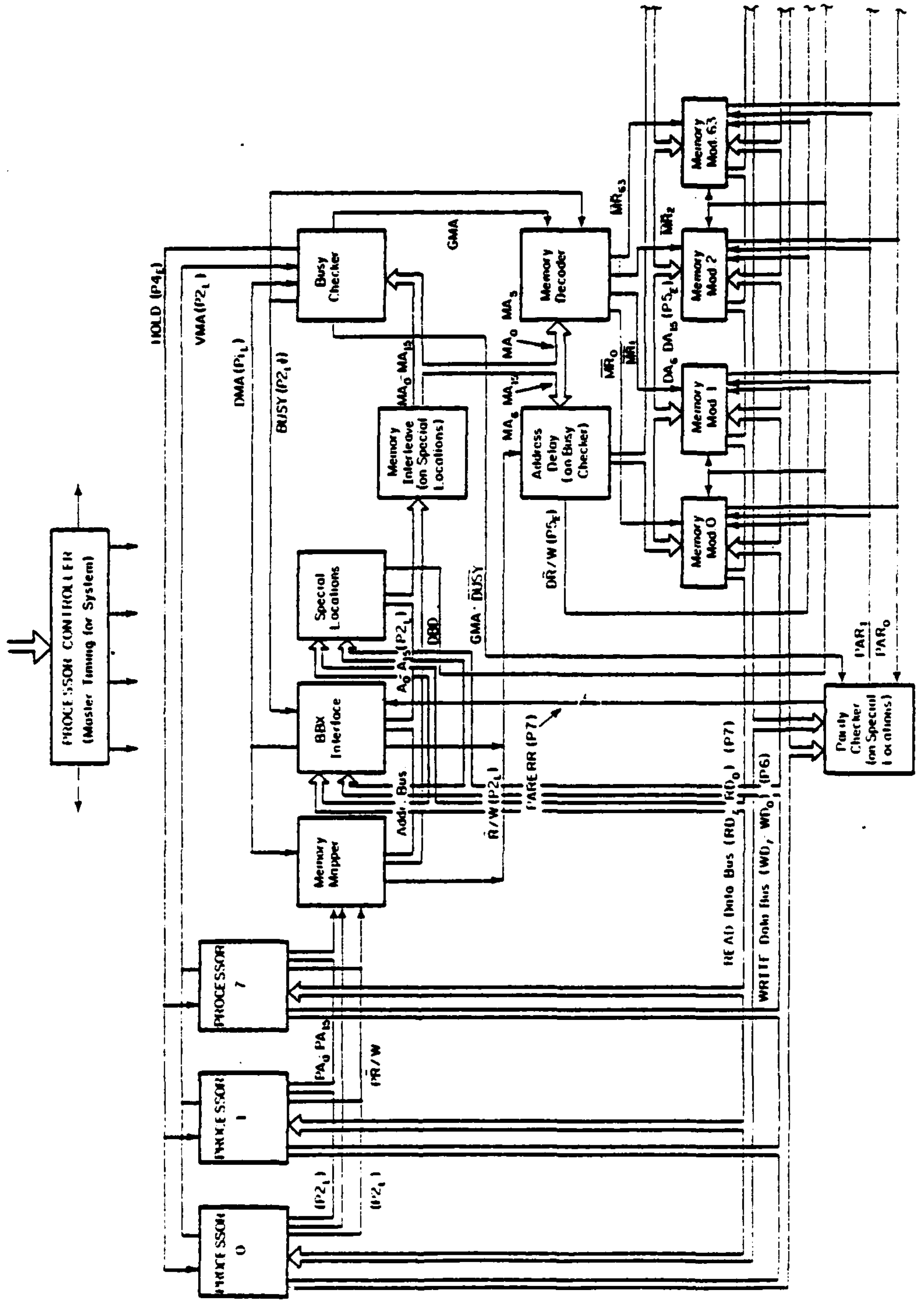


Figure 2.10 The AMP-1 Multiprocessor System

clock times. It also allows reconfiguration of the memory system as 32 modules of 2K bytes each, 16 modules of 4K bytes each, and so forth down to 1 module of the entire 64K bytes.

A memory interleaving plug is provided to allow an arbitrary selection of the level of address interleaving. By selecting different sets of address bits as the memory module number, one can study uninterleaved, two-way interleaved, up to fully interleaved addressing.

The BBX interface connects this system to a DEC System 10 computer. The DEC-10 can read and write any location in memory and can start, stop, and reset arbitrary combinations of processors. It can also be interrupted for message passing from the processors.

A set of special memory-mapped locations are designed and implemented to provide the test-and-set function for critical sections in the programs, interrupt indicator and status message box to the DEC-10, and other functions not otherwise available.

The memory mapper allows each processor to have a small amount (256 bytes) of logically local memory. Local memory is used as temporary working storage and to store enough of the processor state to permit convenient reentrant programming so that the processors can share the same code.

Finally, a hardware monitor (not shown in Figure 2.10) has been designed and built to collect performance information for programs running on the system. Counters in the monitor can be clocked by wiring in chosen system event signals. They can be individually reset and their contents can be latched for reading at any given point by special instructions inserted in the program code.

2.7.2 Software and Computation Decomposition

Software for this system is written in Motorola assembly language. The programs are assembled through a cross-assembler resident in DEC-10.

In order to program a multiprocessor system like AMP-1, one must decompose the work to be done so as to exploit inherent parallelism. Because of the lack of system software support on AMP-1, computation decompositions have to be done completely by the programmer and the desired scheduling is explicitly programmed. Furthermore, because of reliability and portability considerations, it is preferable to code programs with no knowledge of the number of processors available in the system.

Based on the above considerations, a segment of program code, called the job queue, is used to store all jobs waiting to be processed. The jobs are identified in the job queue in terms of a small number of parameters, and the job queue itself is treated as a mutual-exclusion critical section.

The jobs in the job queue may have precedence relations among them. Thus any needed program code is provided at the beginning and the end of job program code to che^k for satisfaction of precedence requirements and indicate completion prior to starting or finishing the job, respectively.

Scheduling is then accomplished simply by having an idle processor interrogate the queue for its next job. When a processor finds an empty job queue, it halts. The computation is complete when all the processors have halted. Other schemes could, of course, be programmed.

2.8 Matrix Multiplication as An Experimental Check of the Model and Two Models for Incorporating Imperfect Job Sharing

A matrix multiplication program, MXMC, has been written and run on the AMP-1 multiprocessor system described in the last section. Matrix multiplication was chosen to focus on resource contention overhead only, since it has a large number of independent jobs with no precedence constraints among them.

The program multiplies two 32x32 matrices and stores the product matrix into a third area. The program computation was divided into independent jobs for scheduling by the job queue. Each job represents a single inner product calculation. We therefore have 1024 independent and identical jobs in the job pool of the program. A mutual-exclusion critical section is used to control the access to the job queue. Only one processor at a time is allowed to fetch a job so no two or more processors would get the same job. Fully-interleaved addressing with all 64 memory modules was used since it tends to distribute memory requests among modules more evenly and dependently, and thus is preferred for checking the model. Data on the effect of address interleaving on the performance of MXMC is presented in the Appendix.

Some pertinent data is given in Table 2.2. The data was measured when a single processor was used and thus are interference-free values.

The critical section length is noninteger since it is the mean value of two different internal paths (one path to index through rows and the other through columns) weighted by their corresponding frequency of occurence in the trace. Likewise, job length is computed as an average value. The fluctuation in the execution time of inner-product jobs comes from the floating-point arithmetic subroutines which are data-dependent (requiring a variable number of normalization shifts). The lengthy inner-product calculation time is due to the 5-byte floating point number format used for each matrix element. For example, the floating point multiplication routine requires an average of 4,669 cycles. Finally, for modeling the critical section as a software resource, the semaphore retry delay, d , is the time required to reaccess the guarding semaphore (during which a software counter for measurement purpose is incremented).

The measured run time data using fully-interleaved addressing are shown in Table 2.3. All times are given in processor cycles. It can be observed that the values predicted by the model (equations 2.1 through 2.3 with resource cycle times replaced by their effective values) are extremely close to the measured data. However, since the job execution time is so large compared to the critical section length, the effect of modeling critical-section access conflict (by equations 2.8 through 2.12) is insignificant (see predicted values of Table 2.3(b).). Nevertheless, this insignificance is consistent with measured data, since the

Table 2.2

Matrix Multiplication Program Data

memory request rate	ψ = 0.7917 requests/cycle

	in processor cycles
initialization cost	34.0
average critical section length	49.34
average inner-product job length	159432.486
semaphore retry delay	18.0

Table 2.3

Matrix Multiplication Run Time Data

total no. of	measured	predicted	predicted time
processors	time	time	measured time
1	163309473	163309473.0	1.000000000
2	82010795	82055137.6	1.000540690
3	55066514	54971038.9	0.998266186
4	41496157	41429493.8	0.998393510
5	33378226	33304965.8	0.997805151
6	28068356	27888942.8	0.993607989
7	24217492	24020633.7	0.991871236

(a) Results when only memory is considered as a resource

total no. of	measured	predicted	predicted time
processors	time	time	measured time
1	163309473	163309473.0	1.000000000
2	82010795	82055143.0	1.000540760
3	55066514	54971046.1	0.998266317
4	41496157	41429501.9	0.998393706
5	33378226	33304974.6	0.997805413
6	28068356	27888951.9	0.993608315
7	24217492	24020643.2	0.991871627

(b) Results when both memory and the critical section are considered as resources critical-section accesses rarely conflict in the real runs of our experiment.

Predicting the matrix multiplication execution times using the above models basically assumes a perfect sharing of the computation work load R_0 among the processors. For comparison, the necessary run time for matrix multiplication is predicted by using two other mëthods: a simple deterministic model and a renewal-theory model. Both of these models explicitly take into consideration the effect of imperfect sharing of R_0 at the end of the computation, which has not been previously modeled.

The simple deterministic model ignores memory interference and treats job execution time and critical section length as constants. The run time of matrix multiplication is calculated by considering the multiprocessing as a deterministic process. The modeled multiprocessing of a 3 x 3 matrix multiplication by 4 processors is shown in Figure 2.11. The general expression for the run time Tp using p processors can easily be obtained and given as follows:

(1) $J_p \geq (p-1)C_p$

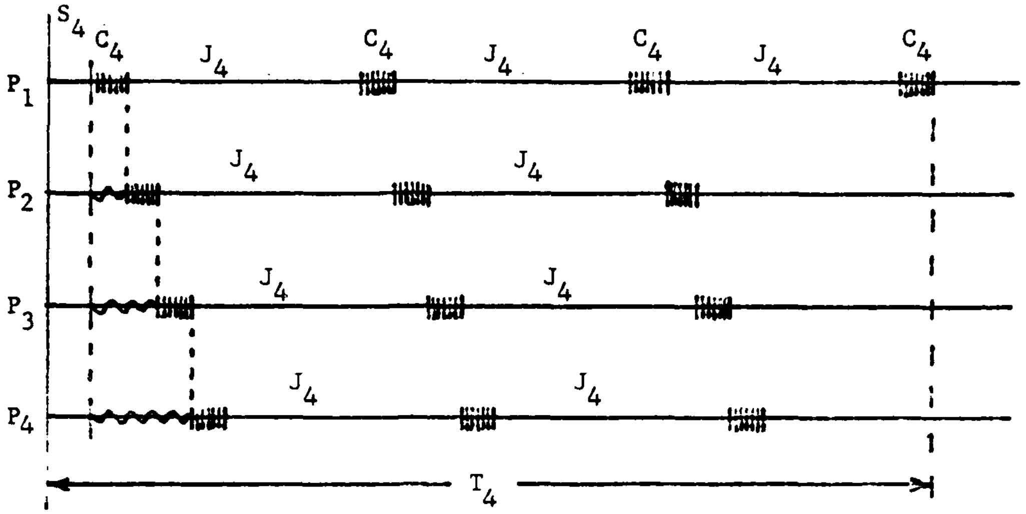
(i) p divides N

$$T_p = S_p + pC_p + \frac{N}{p}(J_p + C_p)$$

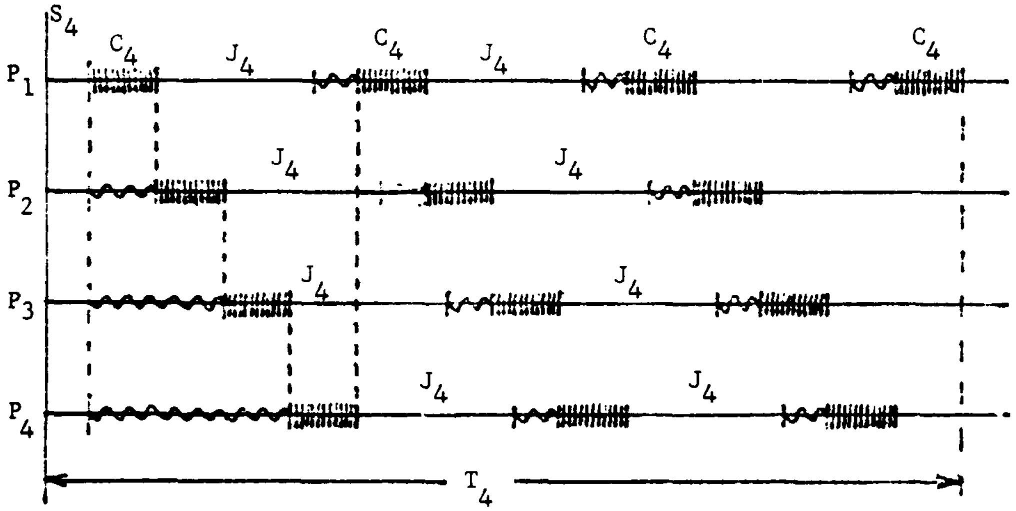
(ii) p does not divide N

$$T_p = S_p + (N - \left\lfloor \frac{N}{p} \right\rfloor p)\, C_p + \left\lceil \frac{N}{p} \right\rceil (J_p + C_p) \qquad (2.23)$$

N = 9



(a) $J_4 \geq 3\ C_4$



(b) $J_4 < 3\ C_4$

Figure 2.11 Run Time Model of 3x3 Matrix Multiplication Using 4 Processors

(2) $J_p < (p-1)C_p$

$$T_p = S_p + (N+p)C_p$$

where N = total number of inner product jobs,

Sp = initialization time,

Jp = job execution time in a p-processor environment,

and Cp = critical section execution time in a p-processor environment.

Run times predicted by the simple deterministic model for the matrix multiplication are shown in Table 2.4(a). Although the predicted values are very close to measured data, the discrepancy does increase as the number of participating processors increases and memory interference gets more severe. Memory interference is neglected in this model.

In order to recognize the effect of memory interference, the interference factor ρ in equation 2.1 can be used to modify the various execution times in the deterministic model. To be more specific, the execution times Sp, Cp, and Jp in different parts of formula 2.23 are multiplied by appropriate values of the interference factor ρ depending on the number of actively participating processors at the particular moment. This results in a new "hybrid model". Predicted values by the hybrid model are shown in Table 2.4(b). The match between measured and predicted times is the best among all models due to the fact that the hybrid model considers both the end effect of job sharing and memory conflicts.

Table 2.4

Matrix Multiplication Run Time Data

total no. of processors	measured time	predicted time	predicted time / measured time
1	163309473	163309473.0	1.000000000
2	82010795	81654827.6	0.995659505
3	55066514	54542867.8	0.990490661
4	41496157	40827578.8	0.983888190
5	33378226	32694005.7	0.979500998
6	28068356	27271623.6	0.971614569
7	24217492	23443961.1	0.968059001

(a) Results from the simple deterministic model

total no. of processors	measured time	predicted time	predicted time / measured time
1	163309473	163309473.0	1.000000000
2	82010795	82055187.0	1.000541300
3	55066514	55076838.2	1.000187490
4	41496157	41429643.3	0.998397113
5	33378226	33336852.5	0.998760465
6	28068356	27941986.5	0.995497795
7	24217492	24134032.9	0.996553770

(b) Results from the hybrid model

Finally, the execution of matrix multiplication can be viewed from another abstraction. Figure 2.12 (a) depicts the execution of N inner product jobs by p processors with job times drawn independently from the same distribution. If we use T_{ij} to represent the execution time of processor i´s jth job, k_i the total number of jobs executed by processor i, and t_i the time instant when processor i finishes and halts, we get the following se. of equations:

$$\sum_{j=1}^{k_1} T_{1j} = t_1$$

$$\sum_{j=1}^{k_2} T_{2j} = t_2$$

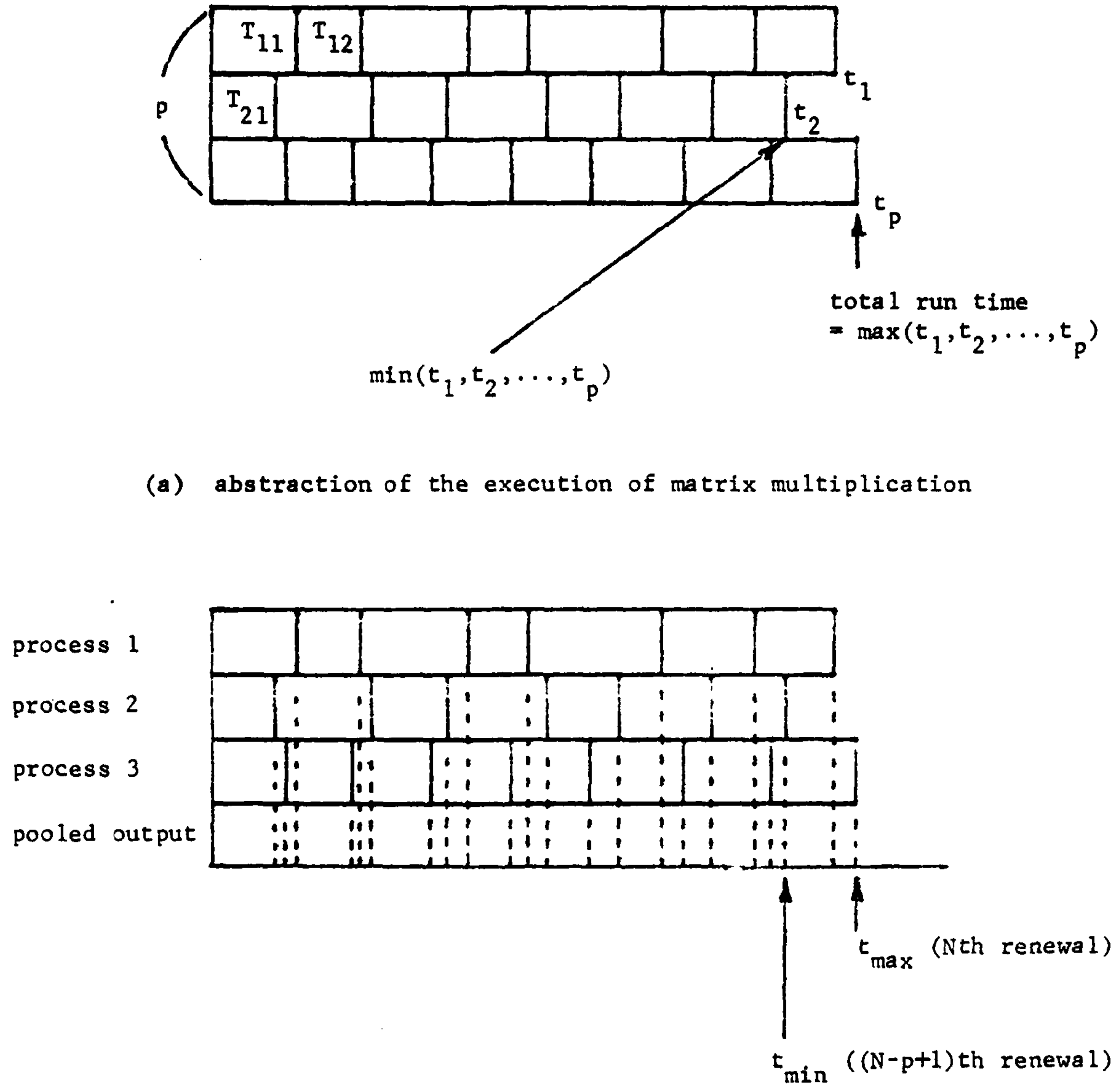
$$\vdots$$

$$\sum_{j=1}^{k_p} T_{pj} = t_p$$

$$\sum_{i=1}^{p} k_i = N$$

$$\text{and} \quad \sum_{j=1}^{k_i-1} T_{ij} \le \min(t_1,t_2, \ldots , t_p) \quad , \quad i=1,2, \ldots ,p \quad . \qquad (2.24)$$

The last equation in (2.24) is due to the way in which jobs are scheduled by the job queue. It is desired to solve for the t_i´s given N and p with the T_{ij}´s characterized by a job time distribution (or at least its first two moments). As indicated in Figure 2.12(a), the maximum of all the t_i´s is the run time of the total computation. This



(a) abstraction of the execution of matrix multiplication

(b) superposition of renewal processes

Figure 2.12 The Renewal-Theory Model for the Execution of Matrix Multiplication

set of equations, if solvable, could be difficult to solve.

However, since the job times are random variables drawn independently from the same distributon, the execution history of each processor can be viewed as the initial segment of a renewal process [Cox62] if the time instants at which jobs are finished are interpreted as renewals.

For convenience we use tmin and tmax to represent the minimum and maximum, respectively, of all t_i's. Results are known for the pooled output of several renewal processes [Cox62]. These results can be used to determine the expected value of tmin (see Figure 2.12(b)). In particular, when N/p is large, we have (Figure 2.12(b) and formula (6) on p.75 of [Cox62])

$$E\{t_{min}\} \approx \frac{N\mu}{p} - \frac{(p-1)(\mu^2+\sigma^2)}{2p\mu} \tag{2.25}$$

where μ and σ^2 are the mean and variance of the job execution time, respectively.

After the time instant tmin, processors start to halt and hence tmax cannot be found by using results for the pooled output of several renewal processes. We can, however, find out the difference between tmax and tmin by evaluating the expected residue life time, by considering the process at the time instant tmin. Again, when N/p is large, this expected residue life time $E\{V_t\}$ can be obtained as (formula (3) on p.64 of [Cox62])

$$E\{V_t\} \approx \frac{1}{2}\,(\mu + \frac{\sigma^2}{\mu}) \quad . \tag{2.26}$$

Therefore, the expression for the average run time becomes (see Figure 2.12(b))

$$\begin{aligned} E\{t_{max}\} &= E\{t_{min}\} + E\{V_t\} \\ &= \frac{N\mu}{p} + \frac{\mu^2+\sigma^2}{2p\mu} \quad . \end{aligned} \tag{2.27}$$

Run time predictions based on equation 2.27 and data in Table 2.2 are shown in Table 2.5. Again, the discrepancy between measured and predicted values increases as the number of participating processors increases. If we modify (by multiplication) the mean and variance of the job execution time by the interference factor ρ and ρ^2, respectively, the match between measured and predicted values is again highly improved, as shown in Table 2.5.

It is apparent that the memory interference is by far the dominating degradation factor in this computation. Nevertheless, the hybrid model intends to catch both the deterministic structure of the computation and the effect of the memory interference. It does yield the best result among all these models. For general computations this approach is expected to be the most practical and satisfactory. In fact, the algorithm proposed in section 4.2 assumes exactly the same flavor.

On the other hand, the renewal-theory model also catches the end

Table 2.5

Matrix Multiplication Run Time Data

total no. of processors	measured time	predicted time	predicted time / measured time
1	163309473	163389216.0	1.000488300
2	82010795	81694649.8	0.996145078
3	55066514	54463127.6	0.989042590
4	41496157	40847366.6	0.984365048
5	33378226	32677909.9	0.979018775
6	28068356	27231605.5	0.970188831
7	24217492	23341388.0	0.963823506

(a) Results based on renewal theory
ignoring memory interference

total no. of processors	measured time	predicted time	predicted time / measured time
1	163309473	163389216.0	1.000488300
2	82010795	82095204.5	1.001029250
3	55066514	54997880.5	0.998753625
4	41496157	41449722.9	0.998881002
5	33378226	33321227.5	0.998292344
6	28068356	27902559.6	0.994093119
7	24217492	24032361.4	0.992355500

(b) Results based on renewal theory
modified by memory interference

effect of job sharing and the effect of memory interference. It differs from the deterministic model due to probabilistic job lengths (vs. average values used in the deterministic model). Further, for data reported in Table 2.5(b), the interference factor ρ was not appropriately adjusted during the V_t period at the end of the computation according to the number of the remaining active processors. If one can get a functional relationship between the expected value of tmin and the expected value of the i-th job finish since tmin as a function of i (e.g. tmax is the (p-1)th job finish since tmin and V_t is the period between them), then one can use appropriate ρ to modify the mean and the variance of the job time in the intervals between successive job finishes. This will definitely improve the predicted values by the renewal-theory model.

A model capable of handling the end effect of job sharing has the potential of dealing with the precedence structures of general computations. This point needs further research.

CHAPTER 3

Memory Interference in Synchronous Computer Systems

3.1 Introduction

Although the control logic for time-division-multiplexed (TDM) busses is straightforward to design, the bus bandwidth required for TDM busses may be too high in many cases. As an alternative, crossbar switches are used in many small multiprocessor systems to interconnect processors with memory modules. The cost for a crossbar switch in a small system is not too expensive, and the performance is higher [Eme79] compared to TDM busses. For large systems compromises (e.g. the network [Law73] and the delta network [Pat79]), rather than the above two extremes, are usually used. In Chapter 2 we discussed memory interference models for systems with TDM busses, and we discuss in this chapter that problem for the other extreme, systems with crossbars.

Since the crossbar switch does not require explicit clock phase shifts among processors, rather the processors and/or memory modules are actually cycled together. Thus in this chapter we discuss the memory interference problem in such synchronous computer systems. Previous work on the memory interference problem for synchronous computer systems is reviewed in section 1.3.

More specifically, we assume that the multiprocessor system contains N processors and M memory modules. Note that N instead of p, will be used in this chapter for the number of processors in the system in order to conform to the nomenclature used in the synchronous system literature. Neither the processors nor the memory modules need necessarily be identical. A crossbar switch is used as the interconnection network between processors and memory modules (Figure 3.1). All processors present their requests, if any, at the beginning of a memory cycle, conflicts are detected and resolved, and all M memory modules are then cycled together. Again, since the only conflict is 2 or more processors requesting the same memory module, as before without loss of generality, we assume memory access time is equal to memory cycle time here.

Chang, Kuck and Lawrie[CKL77] proposed four dependency classes (Figure 3.2) for the address streams produced by processors. The choice of the appropriate class depends on program structure and the machine architecture on which the programs are run. A dependency between any two addresses .n the address stream is defined to be that logical relationship between them such that the second address cannot be accessed (written or read) until the first has been accessed. In Figure 3.2 each node represents a memory address (request for access) and each link a dependency.

Class A corresponds to the address stream generated by a uniprocessor, monoprogrammed machine which has no capability for detecting or bypassing dependencies. Class B corresponds to an array machine (e.g., ILLIAC IV) and is a somewhat restrictive model for a

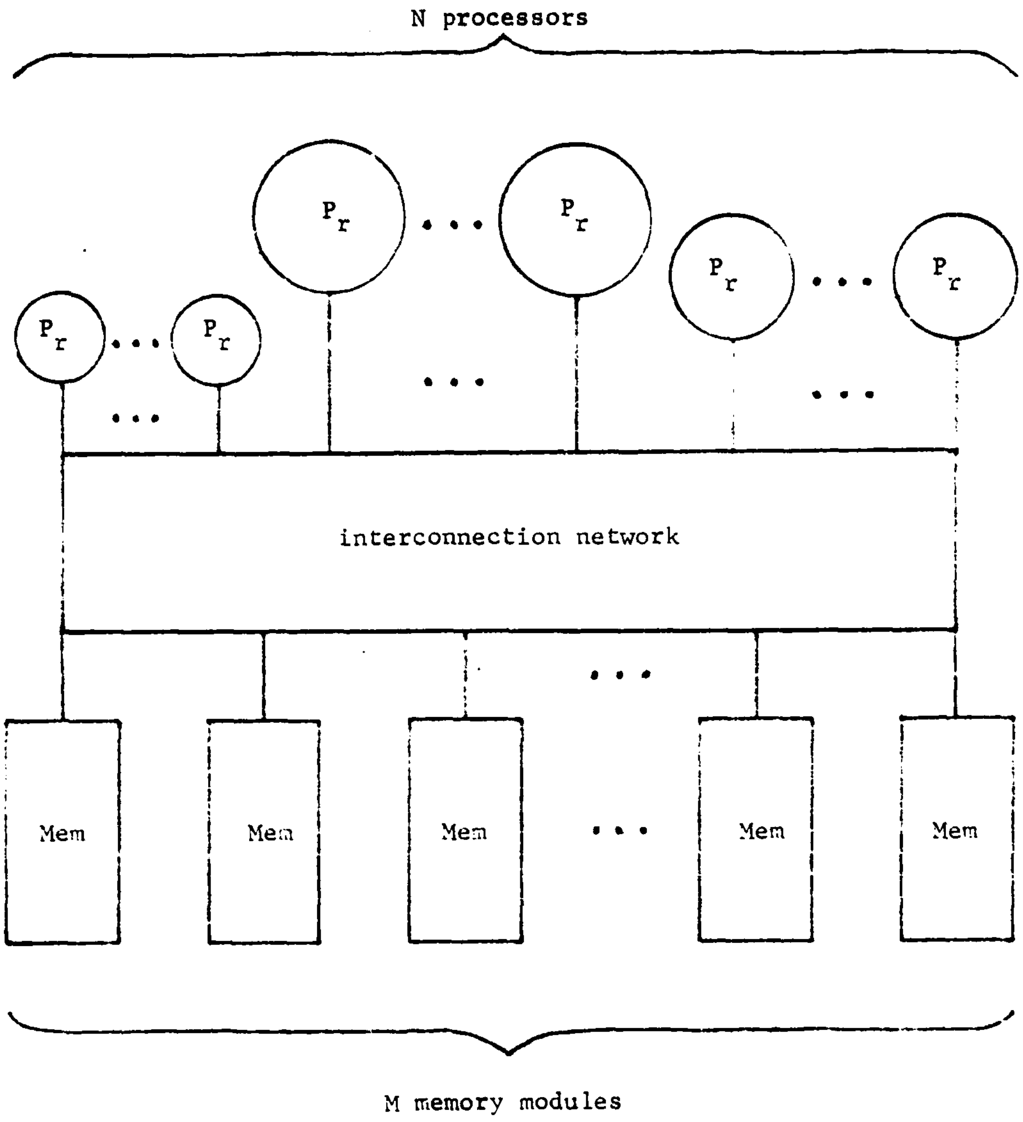
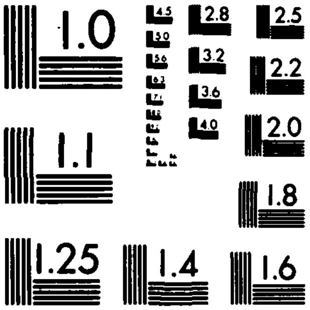


Figure 3.1 A Multiprocessor System

AD-A123 943 PERFORMANCE MODELS FOR MULTIPROCESSOR COMPUTER SYSTEMS 2/2
(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB
D W YEN OCT 80 R-894 N00014-79-C-0424

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

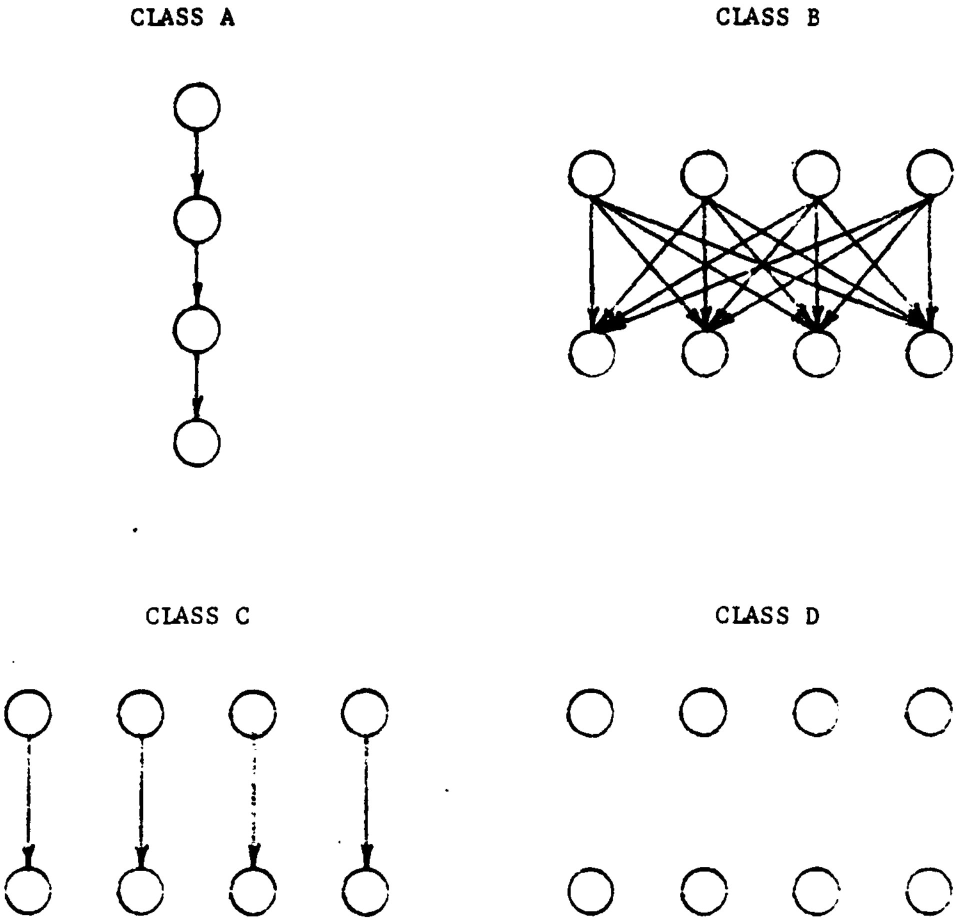


Figure 3.2 Dependency Classes [CKL77]

multifunction machine (e.g., CDC 6600). Address dependency Class C corresponds to a multiprocessor machine and it is this kind of dependency on which we will focus our attention in subsequent sections of this chapter. Class D corresponds to a machine capable of instruction level multiprogramming (from a large number of jobs), or a machine with sufficient lookahead and queueing hardware with respect to memory speed to allow dependencies to be neglected in the model (e.g., IBM 360/91). Various models and simulation studies for the memory interference in these dependency classes of address streams is given in [CKL77].

Two remaining issues related to the applicability of memory interference models are the memory request rate as a function of both processors and memory modules, and the priorities among processors for accessing the memory modules.

Randomness and independence are usually assumed to exist among memory requests produced by processors for the interleaved addressing case. However, a computer system with fully interleaved memory is vulnerable to hardware failures in any single memory module. For certain multiprocessor system applications like weather prediction, atom bomb simulation, structural analysis, and fluid flow dynamics in which the function value of a particular point depends primarily on the values of its immediately neighboring points, interleaved addressing introduces unnecessary interference among processors. (Of course, for the trivial case in which processors deal with disjoint data, interleaved addressing will be responsible for all the data access interference, should it be used.)

Smith[Smi77] proposed a "home memory page placement" scheme which was considered a better data storage scheme for the aforementioned applications. The basic idea is to assign one or more memory modules to each processor in the system and to associate each memory module with at most one processor at a time. Each processor tries to load the data pages in the working set of its process into its own "home memory". Only when overflow occurs do the extra data pages start to migrate to memory modules belonging to other processors. When the home memory assigned to each processor is sufficiently large and higher priority is given to a processor accessing any memory module other than its own home memory, the "home memory page placement" scheme can actually yield better performance (less interference) than interleaved memory.

Since the memory request rate is really a function of the assignment of addresses among the memory modules, for wider applicability a memory interference model should allow module-dependent memory request rates.

Furthermore, the processors in a multiprocessor system may differ either in physical characteristics (e.g., the PDP-11/20's and PDP-11/40's used in C.mmp[OlF78]) or in function (e.g., the arithmetic/ processors and I/O processors in a multiprocessor system). Either reason could make their request rate to the shared memory be different.

A memory interference model, therefore, will be more general if provision for arbitrary request rate distributions from processors and arbitrary request distributions among the memory modules is available.

Finally, what if several processors request a single memory module, thereby causing conflict? In the case of arithmetic/logic processors, which processor´s request is accepted and which request is rejected probably does not matter too much. The rejected processors just waste one cycle and need to resubmit their requests in the next cycle.

However, with I/O processors, the situation is different. Because I/O traffic often consists of transfers between main memory and electromechanical peripheral devices which have a sequential rather than a random access character, there is a relatively large time penalty associated with a lost I/O datum. It is conventional in such systems to avoid any loss of I/O data by granting I/O processors memory service priority over arithmetic/logic processors. Although with extra buffering for I/O data transfer and dynamic priority assignment [Pir67], [Str79], the performance degradation suffered by arithmetic/logic processors because of I/O data transfer could be reduced to be fairly insignificant, a static priority discipline is simpler by far to implement. In view of the typically lower transfer rate of I/O devices, the interference of I/O accesses with the arithmetic/logic processors is reasonably small in any case.

In order to insure wide applicability, under the above considerations, the most general model presented in the following sections for memory interference has provisions imbedded in the model to handle both non-uniform request rates and alternative processor priority schemes.

3.2 A General Model for the Memory Interference in Synchronous Computer Systems

Suppose there are N processors and M memory modules in a multiprocessor system. The memory access time and memory cycle time are assumed to be equal and all M memory modules are cycled simultaneously. Similarly, all processors are synchronized with the memory.

The N processors are divided into m categories with n_i processors in each category. Of course, we have

$$\sum_{i=1}^{m} n_i = N \quad . \tag{3.1}$$

In each memory cycle each processor in category i has a static request rate with stationary probab.lity ψ_{ij} of requesting memory module j, where i = 1,2,...,m. Note that

$$\sum_{j=1}^{M} \psi_{ij} \leq 1 \ . \tag{3.2}$$

Such a probability assignment is equivalent to assuming a geometric distribution for the processing time between memory requests from a particular processor. For convenience, we will sometimes refer to memory requests from processors in category i as category-i requests, where i = 1,2,...,m.

We assume here that successive requests from a processor are independently distributed among the memory modules according to some distribution specified by the static request rates. Requests from different processors are also assumed to be independent of each other.

Now, for a processor in category i, i = 1,2,...,m, the average number of cycles needed per program cycle, ρ_i , can be expressed as

$$\rho_i = 1 - \sum_{j=1}^{M} \psi_{ij} + \sum_{j=1}^{M} \psi_{ij} \frac{1}{P_{A_{ij}}} , \qquad i=1,2, \ldots, m , \qquad (3.3)$$

where $P_{A_{ij}}$ is the probability of acceptance for a category-i request to memory module j.

This parameter ρ_i is referred to in the following as the interference factor for processors in category i, because it serves as a multiplicative factor for the interference-free run time.

Due to the interference among memory requests from various processors and the resulting resubmission of previously rejected requests, the memory request rates seen by the memory modules are actually higher than the programmed request rates, or static request rates. This fact can be reflected by the introduction of another parameter α_{ij} , the dynamic category-i request rate for memory module j. The dynamic request rate α_{ij} can be calculated by noticing the contribution of requests to memory module j made by a processor in category i to the average number of cycles needed per programmed cycle for that processor. In other words,

$$\alpha_{ij} = \frac{\psi_{ij} \frac{1}{P_{A_{ij}}}}{\rho_i} = \frac{\psi_{ij} \frac{1}{P_{A_{ij}}}}{1- \sum_{k=1}^{M} \psi_{ik} + \sum_{k=1}^{M} \psi_{ik} \frac{1}{P_{A_{ik}}}} , \qquad \begin{array}{l} i=1,2, \ldots, m , \\ j=1,2, \ldots, M . \end{array} \qquad (3.4)$$

The expression for the probability of acceptance has to be derived with the particular system structure in mind, as does the corresponding expression for systems with TDM busses (equation 1.2). Thus before the expression for the probability of acceptance is derived, it is convenient to give the expression for the memory bandwidth first. We note that α_{ij} as before is the dynamic request rate during the run time for a processor in category i as seen by memory module j. Then $(1-\alpha_{ij})$ becomes the probability that that processor does not address memory module j. The probability that none of the processors in category i addresses memory module j is $(1-\alpha_{ij})^{n_i}$ and hence the probability that at least one processor in some category addresses memory module j is $1-\prod_{i=1}^{m}(1-\alpha_{ij})^{n_i}$. This last probability is actually the probability that memory module j is busy, since each addressed module becomes busy for one cycle. Summing up for all memory modules we get the average number of busy memory modules per memory cycle, which is by definition the memory bandwidth. That is,

$$\text{Bandwidth} = BW = \sum_{j=1}^{M} [1- \prod_{i=1}^{m} (1-\alpha_{ij})^{n_i}] \quad . \tag{3.5}$$

Now the probability of acceptance for a category-i request to memory module j can be obtained by determining the ratio of accepted category-i requests to memory module j versus the total number of category-i requests to memory module j per memory cycle. Since the total number of requests accepted by memory module j per memory cycle, i.e. the probability for memory module j to be busy, contains the contributions from requests in all categories, the total number of accepted category-i requests to memory module j per memory cycle can be found by subtracting other contributions from the total number of requests accepted by memory module j per memory cycle. Hence

$$P_{A_{ij}} = \frac{[1-\prod_{k=1}^{m}(1-\alpha_{kj})^{n_k}] - \sum_{\substack{k=1\\k\neq i}}^{m} n_k\alpha_{kj}P_{A_{kj}}}{n_i\alpha_{ij}} , \qquad (3.6)$$

where i = 1,2,...,m, and j = 1,2,...,M.

One may notice immediately that we do not really have a set of mM independent equations for the mM probabilities of acceptance here. There are only M independent equations in this set, one for each memory module. In fact what we have for each memory module is an equation of conservation of requests. More specifically, as mentioned above, the total number of requests accepted by memory module j per memory cycle (which is equal to the probability of memory module j being busy) consists of contributions from all categories. In other words, for each memory module we have

$$\sum_{i=1}^{m} n_i\alpha_{ij}P_{A_{ij}} = 1 - \prod_{i=1}^{m}(1-\alpha_{ij})^{n_i} , \quad j=1,2, \ldots,M . \qquad (3.7)$$

Unless we have more information about the system to reduce the number of unknowns($P_{A_{ij}}$'s), we need more equations to make the model a complete set of equations. If the former approach is feasible, as for the two special cases presented later in this section, it will usually yield a fairly simple model. However, in general we have to supplement the above model with some other equations.

A particular probability of acceptance is not just a function of the number of processors in its category and their request rate. It also depends on the system's tie-breaking policy when memory conflicts occur.

Note that we have not yet indicated in the model how memory conflicts are to be resolved. Most previously published models do not address this issue explicitly, because by their assumptions of identical processors and identical memory modules with requests uniformly and independently distributed among them, either an equal tie-breaking priority is implied for all processors or the specific tie-breaking policy just does not affect the system performance indices they dealt with.

In our general model, however, non-uniform request rates are allowed. Furthermore a specific processor category can be assigned a particular priority for memory access. Therefore, it is necessary to indicate explicitly the rule for resolving memory conflicts, referred to as the "priority policy" in the sequel.

The priority policy is highly system-dependent. It can be expressed through the relationship among the probability of acceptance, the various dynamic request rates, and the number of processors in each category. For the important special case in which the memory arbiter is unbiased, for example, the equal-priority policy can be expressed by the following relation:

$P_{A_{ij}}$ = probability[given that one processor in category i references memory module j, no other processor references memory module j]

+ $\frac{1}{2}$ probability[given that one processor in category i references memory module j, one other processor also references memory module j]

+ $\frac{1}{3}$ probability[given that one processor in category i references memory module j, two other processors in the system also reference memory module j]

+ ...

+ $\frac{1}{N}$ probability[given that one processor in category i references memory module j, all other processors in the system also reference memory module j]

$$= \prod_{k=1}^{m} (1-\alpha_{kj})^{r_k} + \frac{1}{2} \sum_{k=1}^{m} r_k \alpha_{kj} (1-\alpha_{kj})^{r_k - 1} \prod_{\substack{h=1 \\ h\neq k}}^{m} (1-\alpha_{hj})^{r_h}$$

$$+ \frac{1}{3} \Big[\sum_{k=1}^{m} \sum_{h=k+1}^{m} r_k \alpha_{kj} (1-\alpha_{kj})^{r_k - 1} r_h (1-\alpha_{hj})^{r_h - 1} \prod_{\substack{g=1 \\ g\neq k,h}}^{m} (1-\alpha_{gj})^{r_g}$$

$$+ \sum_{k=1}^{m} \frac{r_k(r_k - 1)}{2} \alpha_{kj}^{2} (1-\alpha_{kj})^{r_k - 2} \prod_{\substack{g=1 \\ g\neq k}}^{m} (1-\alpha_{gj})^{r_g} \Big]$$

$$+ \,.......$$

$$+ \frac{1}{N} \prod_{k=1}^{m} \alpha_{kj}^{r_k} \,, \qquad \text{where } r_k = n_k \,,\; k\neq i,\; k\in\{1,2,\ldots,m\} \,,$$

$$\text{and } r_i = n_i - 1 \,. \qquad (3.8)$$

The above relation holds for $i = 1,2,\ldots,m$ and $j = 1,2,\ldots,M$. We also require

$$(1-\alpha_{kj})^{\ell} = 0 \quad \text{whenever } \ell < 0 \quad . \tag{3.9}$$

With proper translation this expression has essentially been derived by Hoogendoorn[Hoo77] for a more restricted case. In fact, if there is only one processor in each category equation 3.8 is reduced to equation (1) in [Hoo77] except for some differences in the definitions of parameters. Furthermore, if all the processors in the system have the same request rate and the requests from any processor in the system are uniformly distributed among all memory modules, both of these equations reduce to equation 1.9.

Equation 3.8 is complicated. Since no special characteristics other than the request rates are associated with the categories, a cleaner expression for the probability of acceptance can be obtained if one classifies each processor as a distinct category. In other words, if we use a_{ij} to represent the dynamic request rate for processor i ($i = 1,2,\ldots,N$) referencing memory module j, then we have an expression for the probability of acceptance Pa_{ij} as

$$P_{A_{ij}} = \prod_{\substack{k=1\\k\neq i}}^{N} (1-a_{kj}) + \frac{1}{2} \sum_{\substack{k=1\\k\neq i}}^{N} a_{kj} \cdot \prod_{\substack{h=1\\h\neq i,k}}^{N} (1-a_{hj})$$

$$+ \frac{1}{3} \sum_{\substack{k=1\\k\neq i}}^{N} a_{kj} \sum_{\substack{h=k+1\\h\neq i}}^{N} a_{hj} \cdot \prod_{\substack{g=1\\g\neq i,k,h}}^{N} (1-a_{gj})$$

$$+ \frac{1}{4} \sum_{\substack{k=1 \\ k\neq i}}^{N} a_{kj} \sum_{\substack{h=k+1 \\ h\neq i}}^{N} a_{hj} \cdot \sum_{\substack{g=h+1 \\ g\neq i}}^{N} a_{gj} \cdot \prod_{\substack{\ell=1 \\ \ell\neq i,k,h,g}}^{N} (1-a_{\ell j}) \qquad (3.8')$$

$$\vdots$$

$$+ \frac{1}{N} \prod_{\substack{k=1 \\ k\neq i}}^{N} a_{kj} \quad .$$

However, if unequal priority is assigned to the processor categories, equation 3.8 no longer applies. As an example, if we assume a fixed category-priority scheme, i.e. processors in category 1 have the highest priority, processors in category 2 have the second highest, and so on, then processors in category 1 only compete among themselves and processors with lower priority have a chance only if none of the processors with higher priority accesses the particular memory module. $P_{A_{ij}}$ for category priority is defined by the following set of m equations:

$$P_{A_{1j}} = \frac{1-(1-\alpha_{ij})^{n_1}}{n_1\alpha_{ij}} \quad , \qquad j=1,2,\ldots,M,$$

$$P_{A_{2j}} = (1-\alpha_{ij})^{n_1} \frac{1-(1-\alpha_{2j})^{n_2}}{n_2\alpha_{2j}} \quad , \qquad j=1,2....,M,$$

$$\cdots\cdots\cdots\cdot$$

$$P_{A_{mj}} = \left(\prod_{k=1}^{m-1} (1-\alpha_{kj})^{n_k} \right) \frac{1-(1-\alpha_{mj})^{n_m}}{n_m\alpha_{mj}} \quad , \qquad j=1,2,\ldots,M. \qquad (3.10)$$

It can easily be shown that equation 3.10 is compatible with equation 3.7.

Finally, as mentioned above, we note that in many important cases it is possible to use the information regarding the priority policy to reduce the unknowns in equation 3.6 or 3.7 directly (so that the number of unknowns is equal to the number of independent equations there) without resort to incorporating explicit priority relations as in equations 3.8 and 3.10. Two cases follow by way of example:

Special Case I:

We assume that there are N identical processors and M identical memory modules in this case. Memory requests from a processor are uniformly and independently distributed among all modules. Memory arbitration is unbiased.

In this case there is only one category of processors. Memory request rates, static or dynamic, for all processors are the same and all processors have the same probability of acceptance.

Hence by symmetry we have

$$\psi = \sum_{j=1}^{M} \psi_{1j} = M\psi_{1j} \quad , \tag{3.11}$$

$$\alpha = \sum_{j=1}^{M} \alpha_{ij} = M\alpha_{ij} \quad ,$$

$$\text{and} \quad P_A = P_{A_{1j}} \quad ,$$

The memory interference model is thus reduced to

$$\rho = 1 - \psi + \psi \frac{1}{P_A} \tag{3.12a}$$

$$\alpha = \frac{\psi \frac{1}{P_A}}{1-\psi+\psi \frac{1}{P_A}} = \frac{1}{1+P_A(\frac{1}{\psi} -1)} \tag{3.12b}$$

$$BW = M[1-(1- \frac{\alpha}{M})^{N}] \tag{3.12c}$$

$$\text{and} \quad P_A = \frac{BW}{N\alpha} \; . \tag{3.12d}$$

Note that equations 3.12a and 3.12b are exactly the same as those in Emer´s model. The expression for the probability of acceptance, however, is different due to simultaneous multiple requests rather than the time phased (TDM) requests of Emer´s model.

Special Case II:

Now suppose there are two categories of processors in the system. One category consists of n_p identical arithmetic/logic processors with memory request rate ψ_p. The other category consists of an I/O processor or channel which has a memory request rate ψ_c when operating. The parameter n_c for the channel in equations 3.5 and 3.6 is equal to 1 while the channel is active and 0 otherwise. There are M identical memory modules in the system with requests from any processor or channel, again, uniformly and independently distributed among them. As discussed in section 3.1 a higher memory accessing priority is given to the channel.

Again, by symmetry we have for the arithmetic/logic processors:

$$\psi_p = \sum_{j=1}^{M} \psi_{pj} = M\psi_{pj}$$

$$\alpha_p = \sum_{j=1}^{M} \alpha_{pj} = M\alpha_{pj}$$

$$\text{and} \quad P_{A_p} = P_{A_{pj}} \quad . \tag{3.13}$$

Furthermore, we know immediately that the probability of acceptance for channel request, P_{Ac} , is 1. This value can therefore be substituted into equation 3.6 and we obtain one equation for the single unknown P_{Ap} .

The memory interference model for this case thus becomes

$$\rho_p = 1 - \psi_p + \psi_p \frac{1}{P_{A_p}}$$

$$\rho_c = 1 \tag{3.14}$$

$$\alpha_p = \frac{\psi_p \frac{1}{P_{A_p}}}{1-\psi_p+\psi_p \frac{1}{P_{A_p}}} = \frac{1}{1+P_{A_p}(\frac{1}{\psi_p} -1)}$$

$$\alpha_c = \psi_c$$

$$BW = M[1-(1-\frac{\alpha_p}{M})^{n_p}(1-\frac{\psi_c}{M})^{n_c}]$$

$$P_{Ap} = \frac{BW-n_c\psi_c}{n_p\alpha_p}$$

and $P_{Ac} = 1$, where $n_c = 1$ while the channel is active and 0 otherwise.

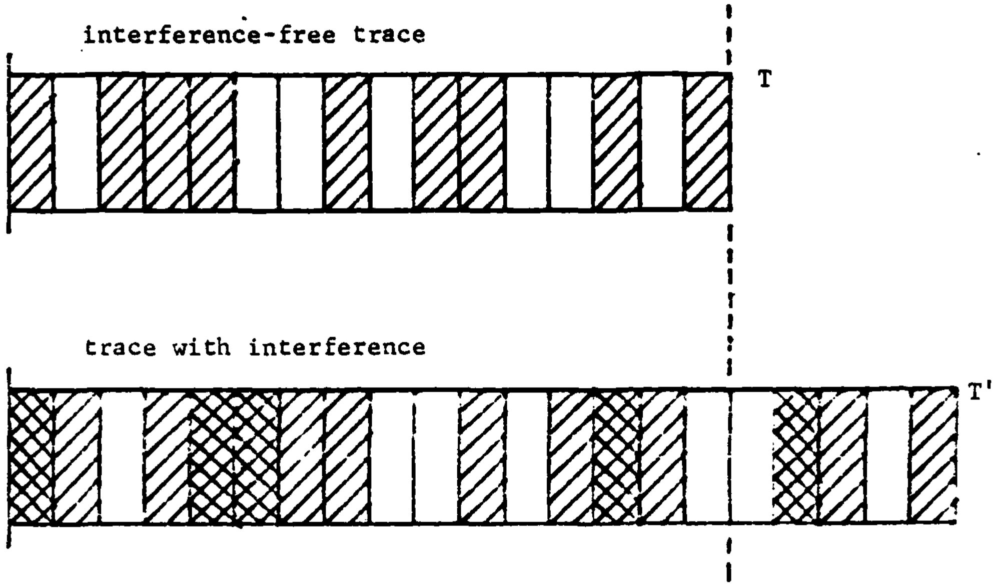
This model is used in a simulation study (to be presented in section 4.3) of the execution of matrix multiplication in a multiprocessor system with virtual memory.

3.3 An Improved Model for the Case of Uniform Access and Equal Priority

As we discussed in section 3.1, for the memory interference problem both the probabilistic approach and the state-space approach have their merits. The general model presented in the previous section is based on a probabilistic argument. For the case of uniform access and equal priority it reduces to equation 1.9 and we know equation 1.9 overestimates bandwidth.

As an attempt to get a more accurate result we derive in this section a model which looks at the problem from a different point of view and employs both probabilistic and state-space concepts. We restrict ourselves in this section to dealing only with the case of uniform access and equal priority. The extension to the case of a fixed category-priority scheme will be discussed in the next section.

In Figure 3.3 a typical cycle trace of a processor is shown. The blank cycles represent those cycles in which the processor is doing internal computation with no generated memory request. The shaded cycle represents a cycle in which the processor submits (or resubmits) a memory request at the beginning and the request is served. The cross-hatched cycle, on the other hand, represents a cycle in which the processor submits (or resubmits) a memory request at the beginning and the request is rejected. For the amount of work which takes the processor T cycles in an interference-free environment, it takes the processor T' cycles ($T' \geq T$) with memory access interference. Note that, in terms of the model parameters discussed previously, the density of the memory-requesting (shaded) part of the T-cycle segment is ψ, while the density of the memory-requesting (shaded and cross-hatched) part of the T'-cycle segment is α. Furthermore, in terms of the interference factor ρ we have $T' = \rho T$. As in Chapter 2, the reciprocal of ρ ($=T/T'$) actually indicates the factor of performance degradation. Since this factor will be used frequently in the rest of this chapter, we give it a special name "the degradation factor" and indicate it by the symbol "f". An interesting expression for f can be obtained by equating the total number of blank cycles in the interference-free trace and in the trace with interference



▨ memory cycle in which memory request is submitted and accepted

▩ memory cycle in which memory request is submitted and rejected

□ internal computation cycle

Figure 3.3 A Cycle Trace of a Processor

in Figure 3.3. In other words, $T(1-\psi) = T'(1-\alpha)$ and we get

$$T' = \frac{1-\psi}{1-\alpha} T$$

$$\text{Hence} \quad f = \frac{T}{T'} = \frac{1-\alpha}{1-\psi} . \qquad (3.15)$$

Since, as ψ is increased, α approaches 1 faster than ψ does ($1 \geq \alpha \geq \psi \geq 0$), f is expected to have a less-than-1 limiting value when $\psi = 1$ even though f=0/0 according to equation 3.15. This limiting value is the amount of performance degradation when $\psi = 1$.

Now for an N-processor multiprocessor system in equilibrium, we can imagine N traces with interference like that in Figure 3.3 being put together. If in those cross-hatched cycles the corresponding processors just become inactive instead of submitting and resubmitting memory requests and getting rejected, the system will essentially appear to be conflict-free. Then the throughput of the system becomes Nf and at any instant there are only Nf active processors in the system. Thus $Nf\psi$ requests are submitted to the memory each cycle and these requests will all be accepted. In other words, the average number of busy memory modules per cycle is $Nf\psi$, which is then the memory bandwidth of the system. In view of Figure 3.3, since each of the N processors has $T\psi$ requests accepted during the T'-cycle period, the average number of accepted requests per cycle, the memory bandwidth, is thus $NT\psi / T' = Nf\psi$.

Now consider Figure 3.4, which depicts the steady processor flow when the multiprocessor system is in equilibrium. It is assumed that

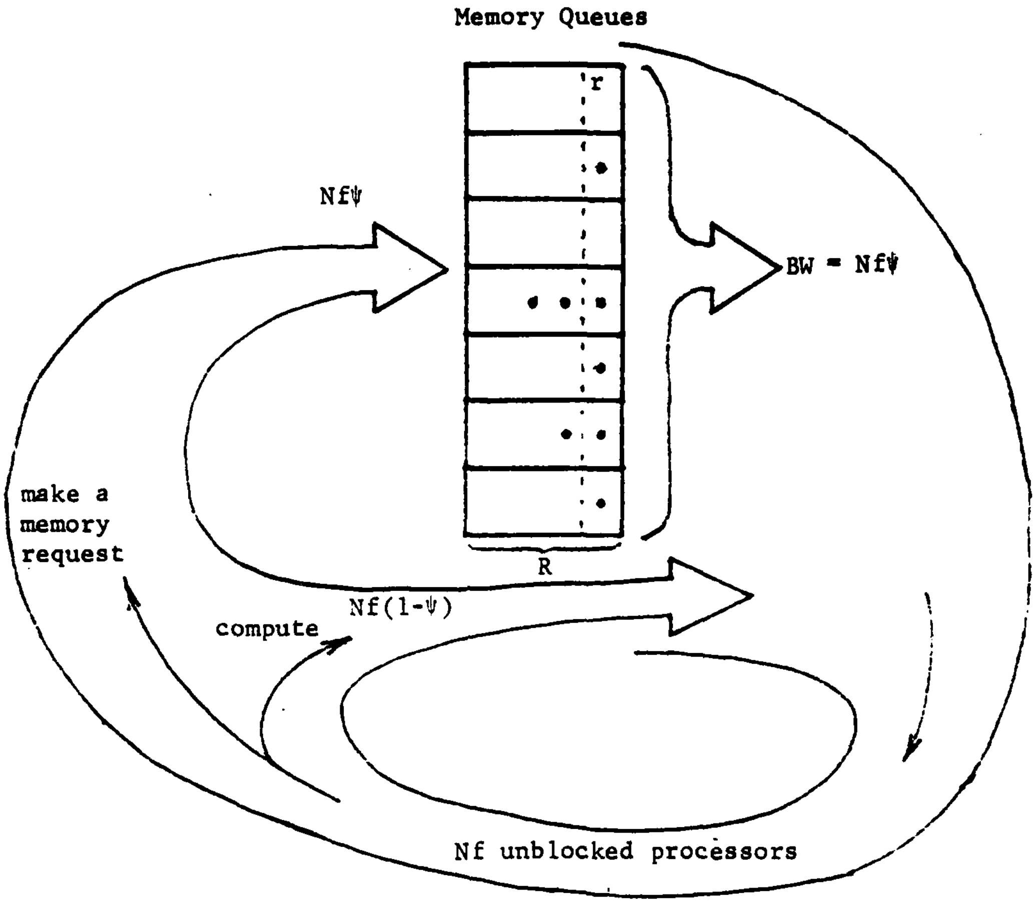


Figure 3.4 A Steady-Flow Model of the Multiprocessor System

there is one queue associated with each memory module. In each cycle each memory module will serve one processor, if any, from the front of the corresponding queue. In Figure 3.4, R is the total number of processors queued in the memory at the end of each memory cycle when the system is in equilibrium, while r is the total number of processors queued in the front of their respective queues. Note that r is also equal to the total number of distinct memory modules in which some processor is queued. Furthermore, r is always smaller than or equal to R. Processors, when not queued, have a memory request rate ψ and the requests, if any, will be presented at the beginning of a memory cycle. The cyclic behavior of the system is thus pictured by the circulation of those processors which are not queued.

Since the average number of busy memory modules per cycle is $Nf\psi$, the average number of processors released from the memory at the end of each memory cycle is also $Nf\psi$. In order to make up the throughput (Nf), there must be $Nf(1-\psi)$ processors each cycle which are doing internal computation and do not deal with the memory. This is indicated by the branch of the flow which bypasses the memory. The average number of processors queued (or retained) in the memory system at the end of each memory cycle, R, can then be found by using the conservation of processors in the system. In other words, $R = N - Nf = N(1-f)$. The $Nf\psi$ processors arriving at the memory system at the beginning of the current memory cycle will face these R processors queued at the memory from last cycle.

We mentioned in section 1.3.1 that the idea of Strecker´s formulation is essentially removing the queued processors from all the memory modules at the end of a memory cycle and reassigning them among all the memory modules at the beginning of the next memory cycle. In the language of Figure 3.4 the total number of processors (or requests) the memory system has to deal with at the beginning of a memory cycle thus becomes $Nf\psi + R$. We note here that

$$\begin{aligned} Nf\psi + R &= Nf\psi + N(1-f) \\ &= N - Nf(1-\psi) \\ &= N - N(1-\alpha) \qquad \text{(from 3.15)} \\ &= N\alpha \quad . \qquad\qquad\qquad\qquad (3.16) \end{aligned}$$

Thus, instead of using $N\alpha$ as the average number of requests submitted to the memory per cycle during the run time, if one interprets α as the probability for a processor making a request during the run time, one obtains equation 1.9. This justifies the statement we made about that equation in section 1.3.1.

However, referring to Figure 3.4, we see that not all the queued processors should be counted as contributing to the total number of busy modules during a cycle. All the processors queued in the same module contribute at most one busy module during a cycle. All the processors queued in the memory but not in the front of their respective queues could be ignored.

Therefore, instead of using all the queued processors, as done by Strecker, we should only take into account those processors queued in the front of their respective queues. The total number of these processors is r. Then, instead of using $Nf\psi + R$ (i.e., $N\alpha$) as the total number of requests randomly and independently assigned to the M modules at the beginning of a memory cycle, we use $Nf\psi+r$ in the exponent of bandwidth formula 1.8. In other words, the total number of busy modules becomes approximately

$$M[1-(1-\frac{1}{M})^{Nf\psi+r}]\ . \tag{3.17}$$

The difficulty here is how to compute the value r. Fortunately although we do not know the exact distribution of the R queued processors in the memory, the total number of memory modules which are going to be busy in the next cycle because of these R queued processors is r. Therefore, as a further approximation, we simply assume that the R queued processors are distributed in the memory as if they were randomly and independently assigned to the M memory modules. Then r can be derived from the formula 1.8. In other words,

$$\begin{aligned} r &\approx M[1-(1-\frac{1}{M})^{R}] \\ &= M[1-(1-\frac{1}{M})^{N(1-f)}] \end{aligned} \tag{3.18}$$

By substituting (3.18) into (3.17) and equating the memory bandwidth Nf to the expression 3.17, we get the following equation for the single unknown f:

$$\begin{aligned} Nf\psi &= M[1-(1-\frac{1}{M})^{Nf\psi+r}] \\ &= M[1-(1-\frac{1}{M})^{Nf\psi+M[1-(1-\frac{1}{M})^{N(1-f)}]}] \end{aligned} \qquad (3.19)$$

Equation 3.19 can be solved for f by iteration using Newton´s method or one of its variants. An IMSL subroutine ZSYSTM based on Brown´s method[Bro69], [BrDe71] is used to solve nonlinear equations of this kind throughout the thesis. From equation 3.15 since ψ is a reasonable initial guess for α, a reasonable initial guess for f is 1.

One may notice that we have been using mean values throughout the above derivation. That is why fractional values appeared in the exponents in several formulas. However, if we view the right-hand side of equation 3.19 as a function and the exponent part of this function its input argument, then the right-hand side of equation 3.19 essentially gives the function value of the average of all feasible input arguments. What we should really look for, though, is the average of the function values of all feasible input arguments. These two may be close, but in general are not necessarily equal. Therefore, instead of settling with equation 3.19 which serves the purpose of providing intuition, we will in the following derive a probabilistic version of equation 3.19 trying to get the real average value that we want.

In order to do so, the meanings of several parameters have to be interpreted differently. Instead of being considered as the "degradation factor", f should now be interpreted as the probability of either doing internal computation or submitting a fresh request for a processor. In other words, f is the probability for a processor to appear in the front of the memory (see Figure 3.4) at the beginning of a memory cycle (before possibly submitting any request). On the other hand, since 1-f is the probability for each of the N processors to be queued in the current memory cycle, the total number of distinct memory modules occupied by queued processors, r, should now be evaluated as

$$r \approx M[1-(1-\frac{1-f}{M})^{N}] \tag{3.20}$$

and r/M now becomes the probability for a memory module to have some processor queued in it.

Referring to Figure 3.4, we can see that the requests which make the memory modules busy come from two sources at the beginning of a memory cycle: the arriving processors and the processors queued in the memory modules. In other words, we consider the system comprising of N processors, M memory queues, and M memory modules. The first two components both contribute to the number of busy modules.

Now the probability that a processor issues a new request is the probability that it is not blocked (=f) times the probability that it generates a request (=ψ). This probability is thus fψ. This request has a probability 1/M of hitting any particular memory module. Hence for a particular memory module not to be hit by any of the arriving processors, the probability is

$$\left(1-\frac{f\psi}{M}\right)^N \quad .$$

Now consider the M memory queues. On average these queues issue r distinct requests per cycle to memory. Thus the probability that a queue makes a request is r/M, or

$$1-\left(1-\frac{1-f}{M}\right)^N \qquad \text{(from equation 3.20)} \quad .$$

This request has a probability 1/M of hitting any particular memory module. Hence the probability for a particular memory module not referenced by any of the M queues is

$$\left(1-\frac{1-\left(1-\frac{1-f}{M}\right)^N}{M}\right)^M \quad .$$

Therefore, the probability for a memory module being referenced and thus being busy, by a processor or a queue, is

$$1-\left(1-\frac{f\psi}{M}\right)^N \left(1-\frac{1-\left(1-\frac{1-f}{M}\right)^N}{M}\right)^M \quad .$$

By equating the bandwidth expression $Nf\psi$ to the total number of busy memory modules using the above probability, we finally obtain the equation we are looking for. That is,

$$Nf\psi = M[1-(1-\frac{f\psi}{M})^{N}(1-\frac{1-(1-\frac{1-f}{M})^{N}}{M})^{M}] \quad . \qquad (3.21)$$

Simulations of the memory interference phenomenon for 4 x 4, 4 x 8, 8 x 4, and 8 x 8 multiprocessor systems have been done on CDC CYBER. Note that when we refer to an N x M multiprocessor system, we mean a multiprocessor system with N processors and M memory modules. The program was written in the simulation language SIMULA and two random number generators were used in the simulation. The first random number generator generates a uniformly-distributed real number in the range of [0,1]. Every cycle a processor, if not queued, will submit a memory request if this random number generator generates a value which is smaller than or equal to ψ. The memory module requested is selected randomly among the M modules according to the number generated by the second random number generator. The processor's request will be queued if not accepted.

Finally, for each system ten values were chosen for ψ (from 0.1 to 1.0) to span its entire feasible range. Each case was run for 45,000 memory cycles. Although regenerative simulation was not implemented, the length of the simulation was sufficiently long and our informal observations indicated that steady state was reached in all cases.

Measured memory bandwidths from the simulations are presented in Figures 3.5 through 3.8. Also shown are the predicted memory bandwidths from various models which allow less-than-one static request rate. The percentage differences of these values compared to the simulation results are shown in Figures 3.9 through 3.12, where the Y-axis value is the difference (analytical - simulation) relative to the simulation result. The actual data for the information contained in above figures are also summarized in Tables 3.1 through 3.4 for reference.

For comparison, we also included a "transient" model in those figures and tables. Notice that when the cyclic flow of Figure 3.4 just starts (at the beginning of the first cycle), no processor is queued and the probability for each processor to make a request in this first cycle is ψ. The resulting memory bandwidth is exactly

$$BW_t = M[1-(1-\frac{\psi}{M})^N] \qquad (3.22)$$

for the first cycle. If one ignores all the rejected requests, then every cycle becomes the first cycle! The memory bandwidth thus obtained is actually the memory bandwidth for the first cycle, not the steady state value.

Note that ψ could be 1, which makes equation 3.22 the same as equation 1.8. Actually, when the request rate $\psi=1$, as long as we assume independence among requests it really does not matter whether the rejected requests are discarded or resubmitted, as far as the memory bandwidth is concerned.

Equation 3.22 is included in the comparisons as Model 1. Model 2 is Strecker and Hoogendoorn's model (equation 1.9), Model 3 is Baskett and Smith's model (equations 1.12 and 1.13), Model 4 is equation 3.19, and

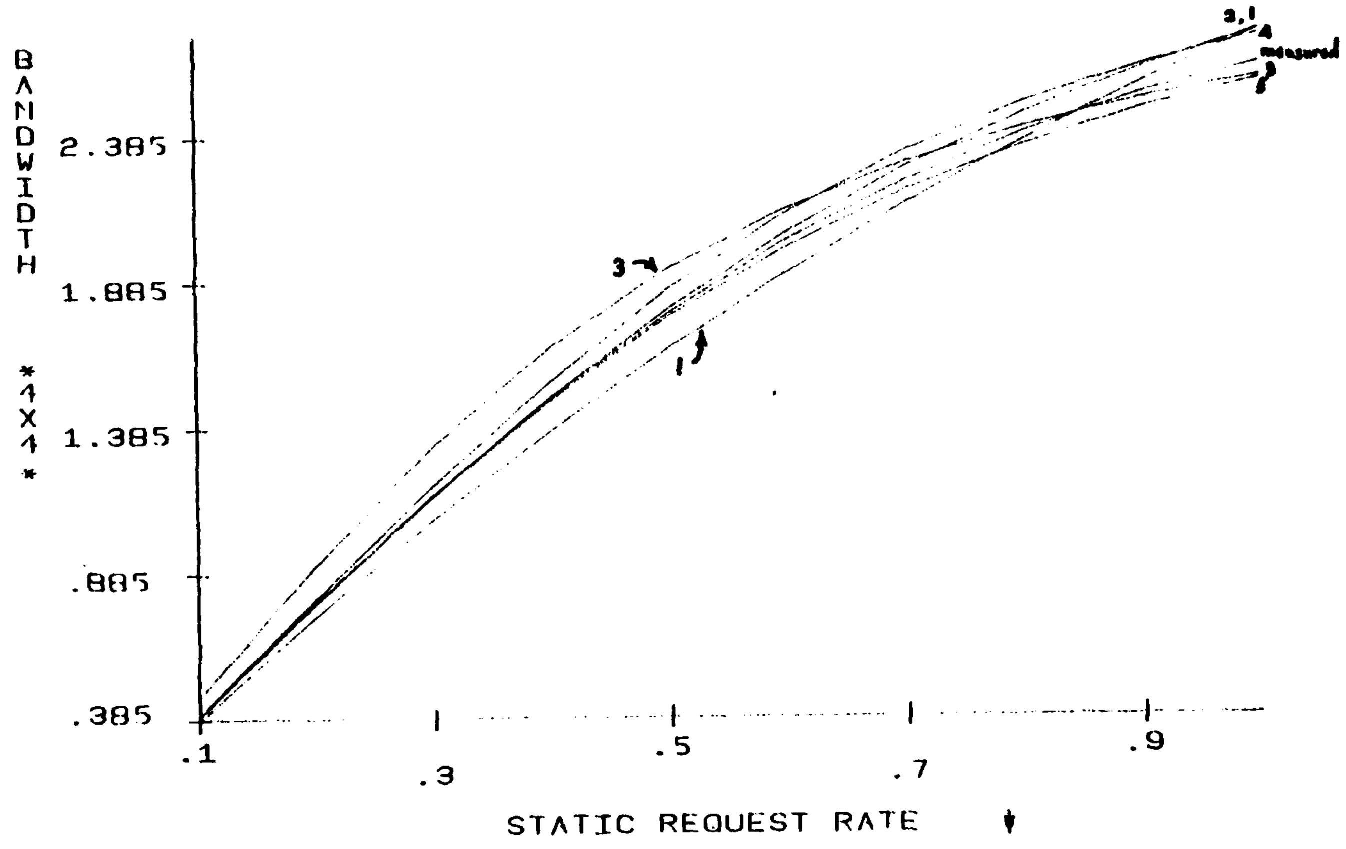


Figure 3.5 Memory Bandwidth vs. Static Request Rate for a 4 x 4 Multiprocessor System

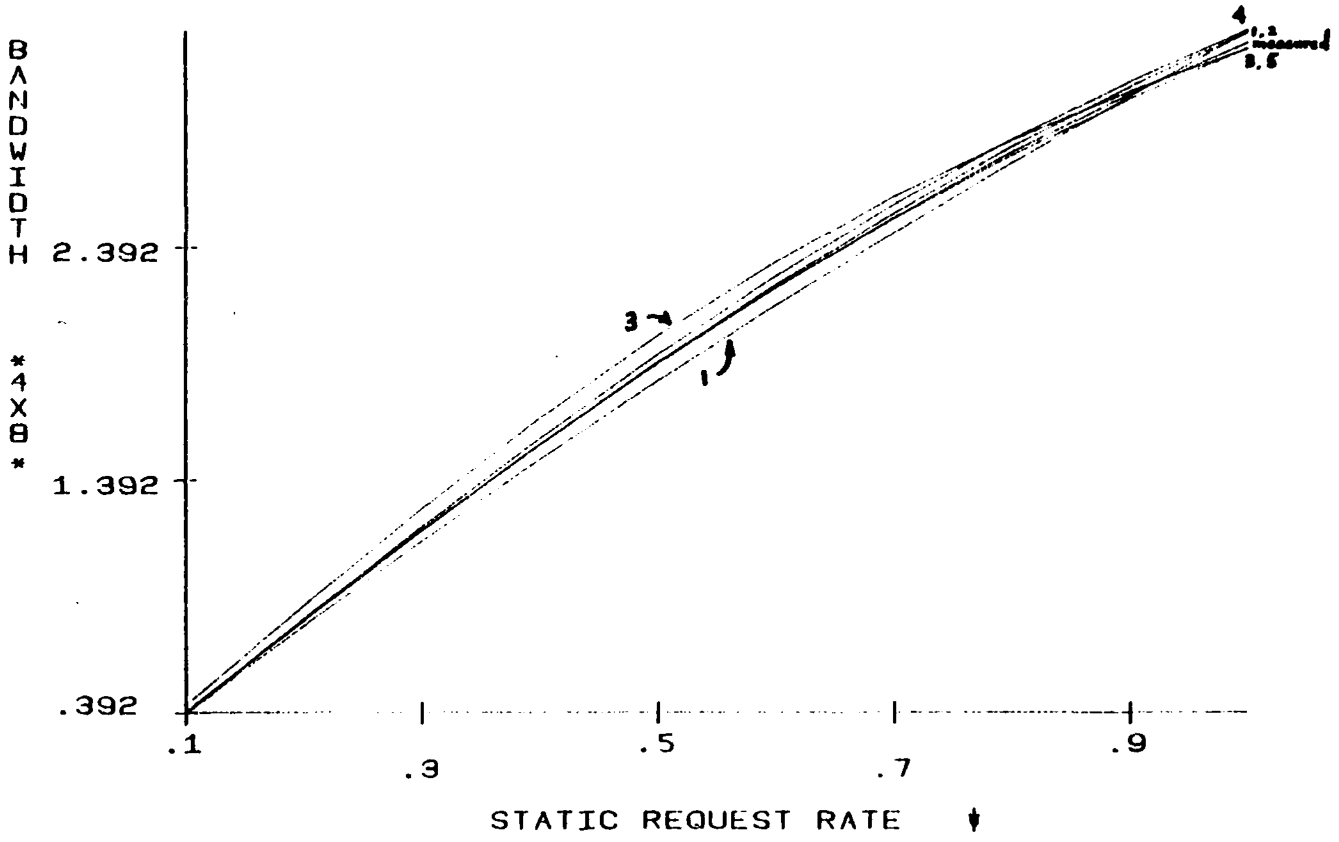


Figure 3.6 Memory Bandwidth vs. Static Request Rate for a 4 x 8 Multiprocessor System

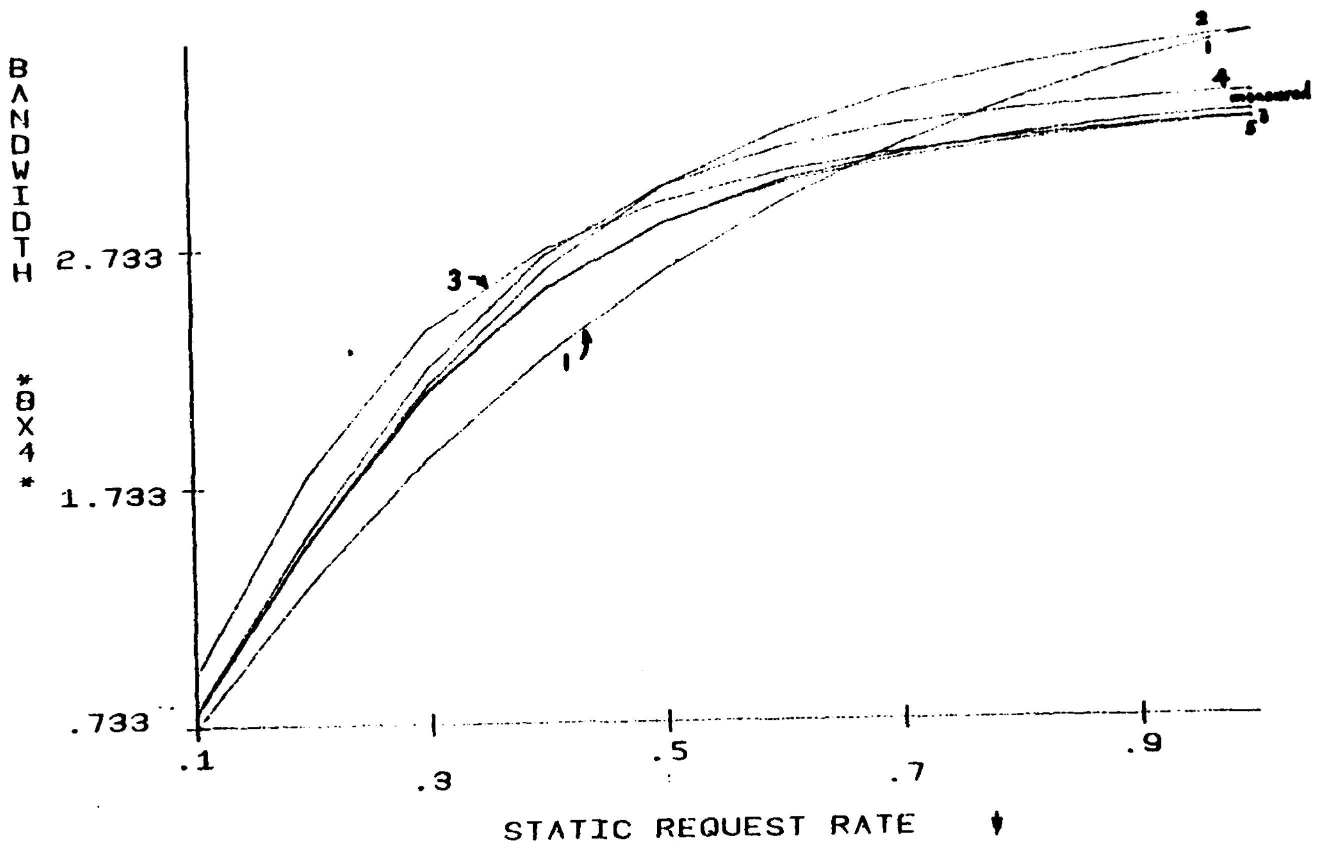


Figure 3.7 Memory Bandwidth vs. Static Request Rate for a 8 x 4 Multiprocessor System

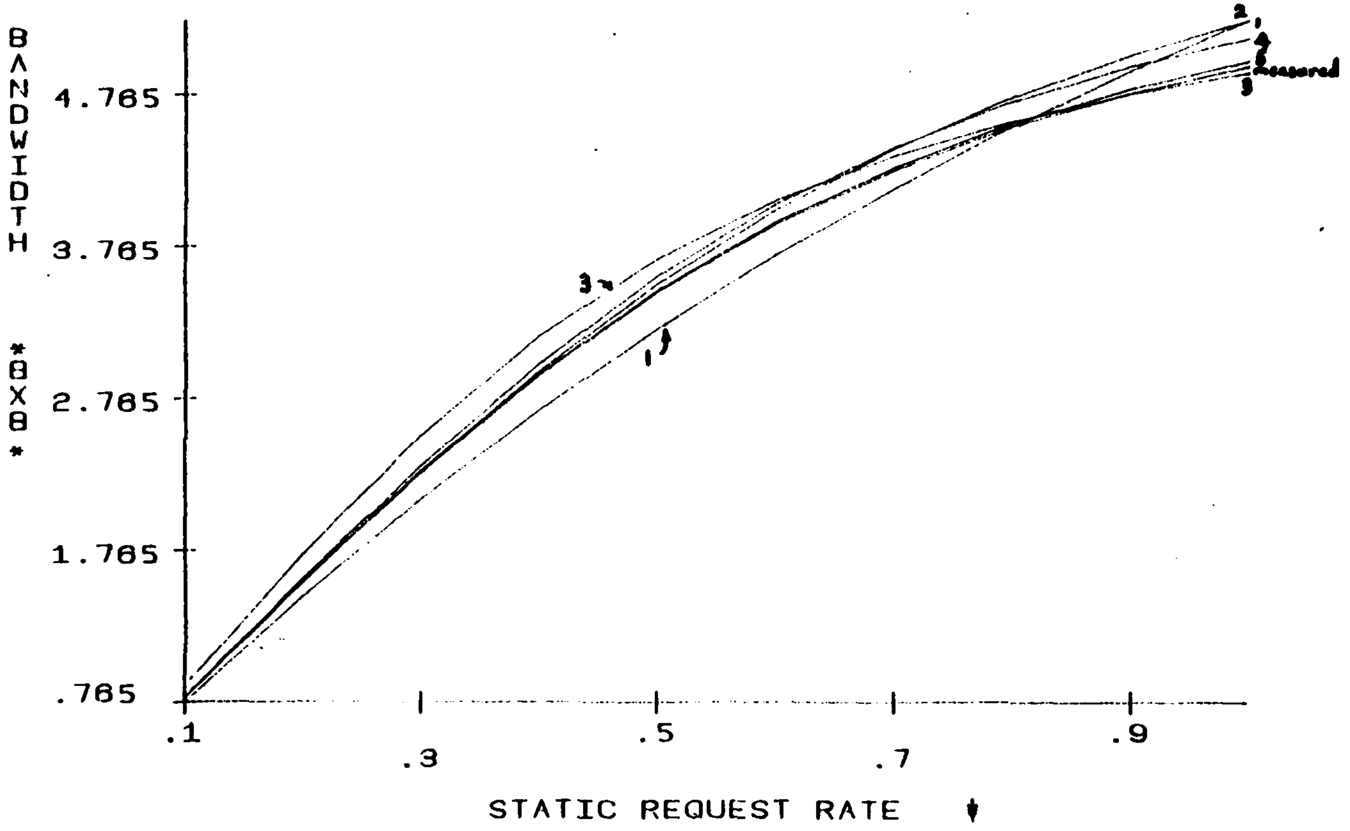


Figure 3.8 Memory Bandwidth vs. Static Request Rate for a 8 x 8 Multiprocessor System

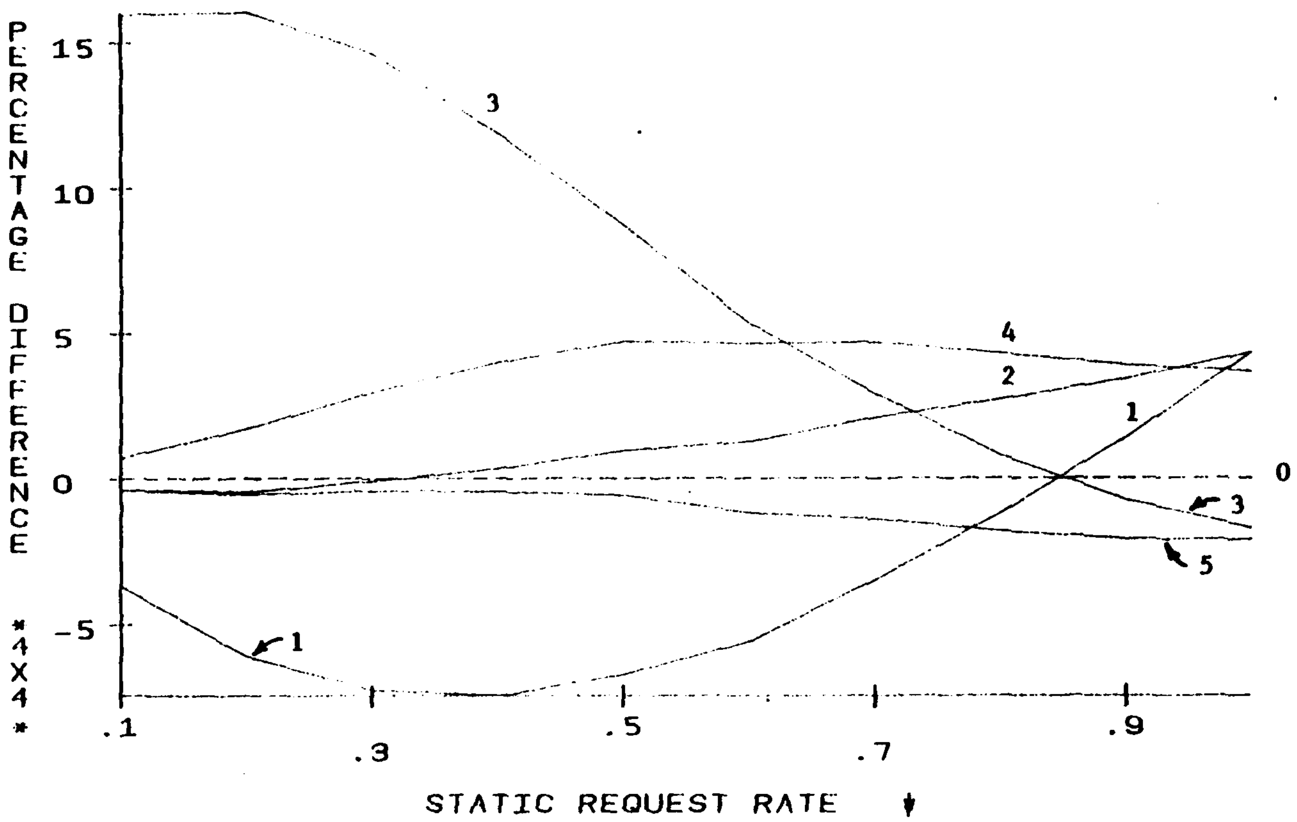


Figure 3.9 Percentage Differences Between Predicted and Simulated Bandwidths for a 4 x 4 Multiprocessor System

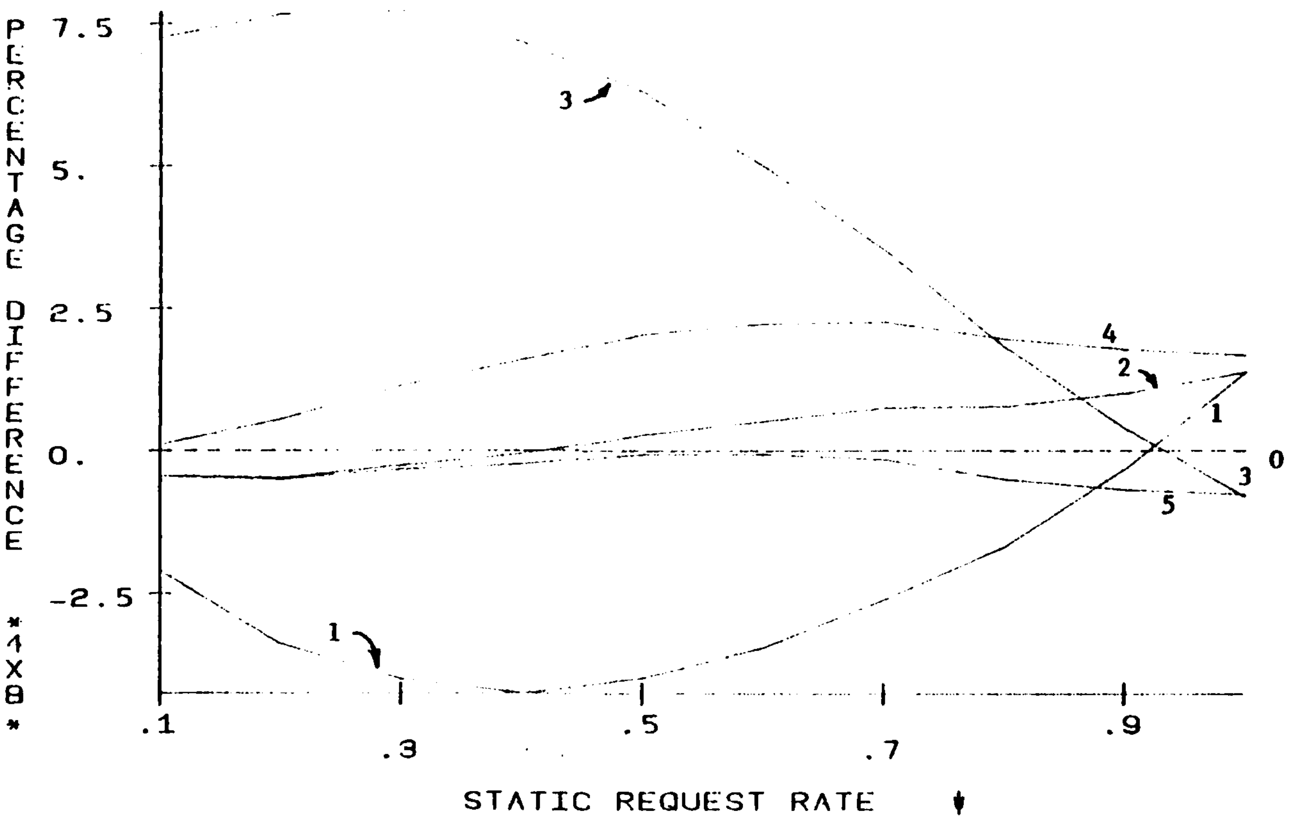


Figure 3.10 Percentage Differences Between Predicted and Simulated Bandwidths for a 4 x 8 Multiprocessor System

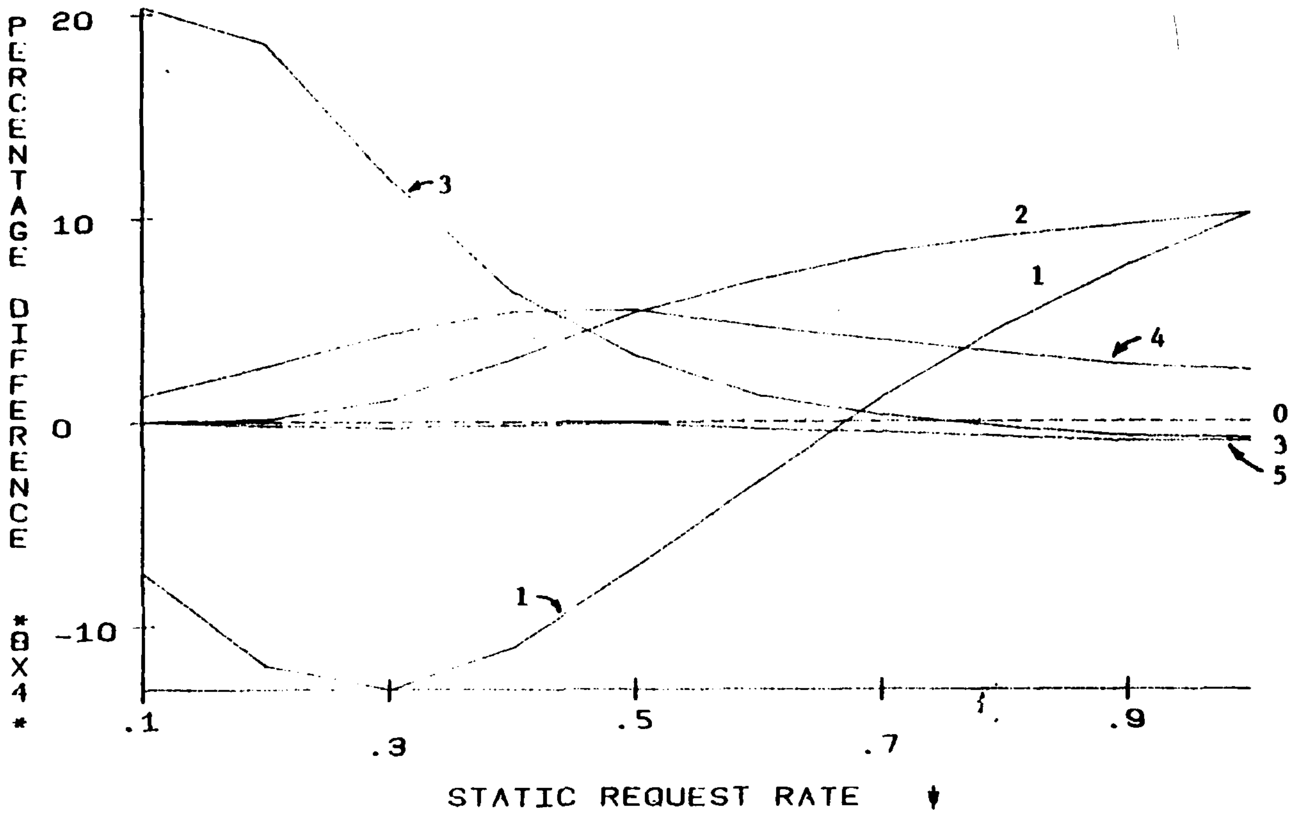


Figure 3.11 Percentage Differences Between Predicted and Simulated Bandwidths for a 8 x 4 Multiprocessor System

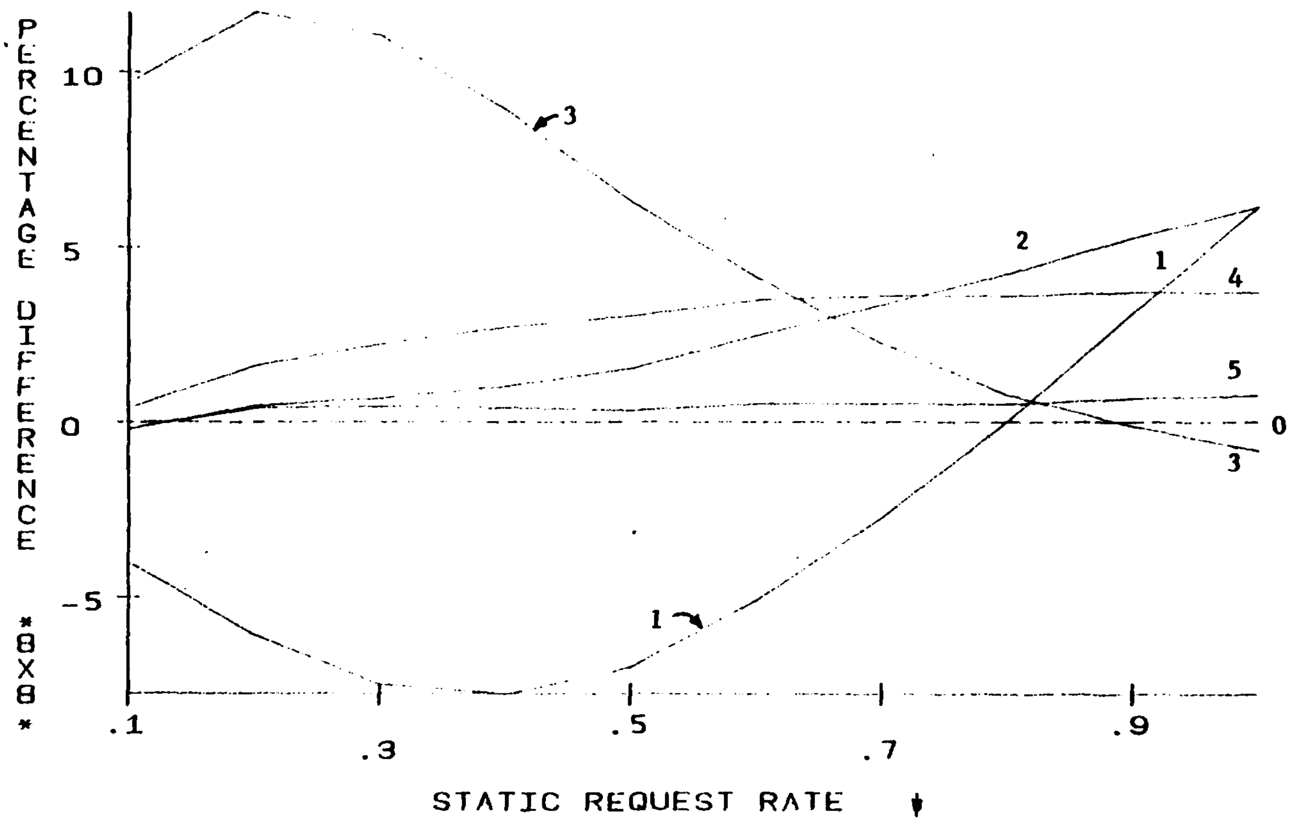


Figure 3.12 Percentage Differences Between Predicted and Simulated Bandwidths for a 8 x 8 Multiprocessor System

Table 3.1
Comparison of Models in Predicted Bandwidth and Percentage Difference with the Simulation Data for a 4 x 4 Multiprocessor system

Model 1 : Equation 3.22
Model 2 : Strecker-Hoogendoorn's model
Model 3 : Baskett-Smith's model
Model 4 : Equation 3.19
Model 5 : Equation 3.21

ψ	SIM BW	BW(1)	BW(2)	BW(3)	BW(4)	BW(5)
.1	.400044	.385248	.398419	.463872	.402930	.398376
.2	.790524	.741975	.786848	.917339	.804155	.786106
.3	1.155800	1.071623	1.154589	1.324702	1.190184	1.150682
.4	1.486711	1.375600	1.491864	1.663215	1.546017	1.479725
.5	1.774783	1.655273	1.791721	1.929536	1.858300	1.764247
.6	2.025400	1.911975	2.051170	2.133500	2.119215	2.001107
.7	2.224368	2.146998	2.271049	2.288949	2.328504	2.192962
.8	2.389902	2.361600	2.454910	2.408427	2.492066	2.346084
.9	2.521055	2.556998	2.607643	2.501626	2.618565	2.467839
1.0	2.621000	2.734375	2.734375	2.575571	2.716585	2.565052

ψ	DIFF(1)%	DIFF(2)%	DIFF(3)%	DIFF(4)%	DIFF(5)%
.1	-3.69848	-.40621	15.95520	.72145	-.41683
.2	-6.14137	-.46506	16.04189	1.72431	-.55892
.3	-7.28297	-.10478	14.61346	2.97492	-.44283
.4	-7.47361	.34660	11.87214	3.98908	-.46991
.5	-6.73376	.95438	8.71955	4.70579	-.59367
.6	-5.60013	1.27235	5.33722	4.63190	-1.19941
.7	-3.47827	2.09860	2.90336	4.68160	-1.41189
.8	-1.18423	2.72010	.77513	4.27480	-1.83346
.9	1.42573	3.43460	-.77066	3.86784	-2.11087
1.0	4.32564	4.32564	-1.73327	3.64688	-2.13460

Table 3.2
Comparison of Models in Predicted Bandwidth and Percentage Difference with the Simulation Data for a 4 x 8 Multiprocessor System

Model 1 : Equation 3.22
Model 2 : Strecker-Hoogendoorn's model
Model 3 : Baskett-Smith's model
Model 4 : Equation 3.19
Model 5 : Equation 3.21

ψ	SIM BW	BW(1)	BW(2)	BW(3)	BW(4)	BW(5)
.1	.401000	.392562	.399230	.430106	.401486	.399220
.2	.797600	.770497	.793727	.858707	.802037	.793559
.3	1.181667	1.134172	1.178596	1.272853	1.195330	1.177722
.4	1.550044	1.483950	1.549128	1.661009	1.574854	1.546357
.5	1.895844	1.820190	1.901199	2.015014	1.934551	1.894567
.6	2.220000	2.143247	2.231598	2.330858	2.269463	2.218428
.7	2.519277	2.453469	2.538221	2.608245	2.576219	2.515359
.8	2.798444	2.751200	2.820096	2.849466	2.853228	2.784256
.9	3.046510	3.036781	3.077259	3.058210	3.100552	3.025374
1.0	3.265200	3.310547	3.310547	3.238644	3.319553	3.240030

ψ	DIFF(1)%	DIFF(2)%	DIFF(3)%	DIFF(4)%	DIFF(5)%
.1	-2.10416	-.44131	7.25829	.12126	-.44380
.2	-3.39808	-.48562	7.66138	.55626	-.50669
.3	-4.01935	-.25990	7.71671	1.15623	-.33382
.4	-4.26401	-.05908	7.15883	1.60062	-.23785
.5	-3.99050	.28245	6.28585	2.04168	-.06736
.6	-3.45735	.52242	4.99362	2.22805	-.07083
.7	-2.61220	.75197	3.53150	2.26024	-.15553
.8	-1.68822	.77371	1.82324	1.95765	-.50700
.9	-.31935	1.00931	.38406	1.77389	-.69379
1.0	1.38879	1.38879	-.81330	1.66461	-.77084

Table 3.3
Comparison of Models in Predicted Bandwidth and Percentage Difference with the Simulation Data for a 8 x 4 Multiprocessor System

Model 1 : Equation 3.22
Model 2 : Strecker-Hoogendoorn's model
Model 3 : Baskett-Smith's model
Model 4 : Equation 3.19
Model 5 : Equation 3.21

ψ	SIM BW	BW(1)	BW(2)	BW(3)	BW(4)	BW(5)
.1	.792000	.733393	.792144	.953279	.802093	.791889
.2	1.529444	1.346318	1.531565	1.813190	1.571086	1.526213
.3	2.136714	1.856153	2.159970	2.392435	2.229436	2.130140
.4	2.561928	2.278131	2.640754	2.725727	2.700363	2.556649
.5	2.826693	2.625564	2.978926	2.917761	2.981554	2.824302
.6	2.998511	2.910038	3.207368	3.035833	3.138008	2.986600
.7	3.104618	3.141596	3.361603	3.113703	3.228153	3.087839
.8	3.178651	3.328911	3.468076	3.168196	3.283874	3.154258
.9	3.232130	3.479437	3.543868	3.208180	3.320736	3.200107
1.0	3.265700	3.599548	3.599548	3.238644	3.346540	3.233196

ψ	DIFF(1)%	DIFF(2)%	DIFF(3)%	DIFF(4)%	DIFF(5)%
.1	-7.39990	.01822	20.36349	1.27438	-.01401
.2	-11.97335	.13865	18.55221	2.72271	-.21123
.3	-13.13050	1.08842	11.96797	4.33948	-.30769
.4	-11.07747	3.07683	6.39356	5.40356	-.20607
.5	-7.11533	5.38554	3.22170	5.47853	-.08450
.6	-2.95057	6.96536	1.24469	4.65220	-.39723
.7	1.19107	8.27751	.29263	3.97908	-.54045
.8	4.72717	9.10526	-.32893	3.31030	-.76739
.9	7.65151	9.64497	-.74100	2.74142	-.99076
1.0	10.22287	10.22287	-.82848	2.47541	-.99532

Table 3.4
Comparison of Models in Predicted Bandwidth and Percentage Difference with the Simulation Data for a 8 x 8 Multiprocessor System

Model 1 : Equation 3.22
Model 2 : Strecker-Hoogendoorn's model
Model 3 : Baskett-Smith's model
Model 4 : Equation 3.19
Model 5 : Equation 3.21

ψ	SIM BW	BW(1)	BW(2)	BW(3)	BW(4)	BW(5)
.1	.797867	.765861	.796300	.875346	.801022	.796246
.2	1.561889	1.466786	1.569215	1.744875	1.587044	1.568211
.3	2.278616	2.107550	2.294067	2.530219	2.329298	2.288494
.4	2.919691	2.692637	2.949465	3.180252	2.998961	2.931501
.5	3.469323	3.226244	3.522089	3.687605	3.574493	3.480439
.6	3.911533	3.712305	4.008629	4.073387	4.047805	3.931603
.7	4.270832	4.154499	4.414196	4.366108	4.424686	4.292944
.8	4.555731	4.556262	4.748819	4.590705	4.719641	4.578774
.9	4.772632	4.920807	5.024110	4.765946	4.949594	4.804582
1.0	4.947100	5.251129	5.251129	4.905190	5.129877	4.984079

ψ	DIFF(1)%	DIFF(2)%	DIFF(3)%	DIFF(4)%	DIFF(5)%
.1	-4.01139	-.19642	9.71072	.39545	-.20316
.2	-6.08900	.46908	11.71568	1.61052	.40474
.3	-7.50744	.67810	11.04191	2.22424	.43351
.4	-7.77666	1.01978	8.92427	2.71500	.40448
.5	-7.00652	1.52092	6.29177	3.03142	.32042
.6	-5.09334	2.48229	4.13787	3.48386	.51309
.7	-2.72390	3.35682	2.23086	3.60245	.51774
.8	.01166	4.23835	.76768	3.59788	.50580
.9	3.10469	5.26916	-.14009	3.70785	.66943
1.0	6.14559	6.14559	-.84716	3.69462	.74749

finally Model 5 is equation 3.21. All these models are summarized in Table 3.5.

We first note from Figures 3.5 through 3.12 that for a good range of the static request rate ψ Model 1 underestimates the memory bandwidth. This is reasonable because contributions from resubmitted requests are totally ignored in Model 1. However, as ψ approaches 1, even Model 1 starts to overestimate. This is because as the memory interference gets more serious processors are more often queued in the same module. Thus in reality many requests from queued processors do not actually increase the total number of busy modules. Since in this region ψ and α only differ slightly, Model 1 overestimates the memory bandwidth even though ψ is used.

On the other hand, since Model 2 is derived based on the assumption that all the queued processors will be assigned randomly and independently to the M modules in the next cycle, from our discussion in section 3.3 it is conjectured that Model 2 always yields an upper bound for the memory bandwidth. This conjecture is not supported by the data presented in above figures and tables, but the deviation is so small that it could be due to the inaccuracies of the simulation results.

Model 3, interestingly enough, is pretty accurate when ψ approaches 1. However, its erratic behavior for smaller ψ makes it a bad model in this range.

The behaviors of Model 4 and 5, on the other hand, are much more stable over the entire range ((0,1]) of ψ than those of the other three

Table 3.5

Memory Interference Models for Synchronous Computer Systems

Model 1: $BW = M[1-(1-\frac{\psi}{M})^{N}]$

Model 2: $BW = M[1-(1-\frac{\alpha}{M})^{N}]$

$$\text{where } \alpha = \frac{1}{1+P_A(\frac{1}{\psi}-1)}$$

$$\text{and } P_A = \frac{M[1-(1-\frac{\alpha}{M})^{N}]}{N\alpha}$$

Model 3: $BW = \frac{M}{2}(2+2L-\frac{1}{M}-\sqrt{(2+2L-\frac{1}{M})^{2}-8L)}$

$$\text{where } L = \frac{-(1+T-\frac{N-1}{M})+\sqrt{(1+T-\frac{N-1}{M})^{2}+4(\frac{N-1}{M})}}{2(\frac{N-1}{N})}$$

$$\text{and } T = \frac{1}{\psi}-1$$

Model 4: $BW = Nf\psi = M[1-(1-\frac{1}{M})^{Nf\psi+M[1-(1-\frac{1}{M})^{N(1-f)}]}]$

Model 5: $BW = Nf\psi = M[1-(1-\frac{f\psi}{M})^{N}(1-\frac{1-(1-\frac{1-f}{M})^{N}}{M})^{M}]$

models. The accuracy of Model 5 (equation 3.21) makes it the best model among all models investigated.

3.4 The Application of the Improved Model in an Environment with Unequal Processor Priorities

One of the strong points of the probabilistic approach is its capability of accommodating priorities without too much trouble. We will show in this section by an example that the model 5 (equation 3.21) presented in the previous section can also be applied to deal with priorities without too much effort.

Again, we assume m categories of processors with n_i processors in each category. The total number of processors in the system is N. There are M identical memory modules in the system with requests from any processor uniformly and independently distributed among them. The memory request rates for processors in the same category are assumed to be the same. Priorities in accessing the memory are associated with the processors' categories. Category 1 has the highest priority, category 2 the second highest, and so forth.

Although conservation of processors should apply to each of the m categories when the system is in steady state, processors in every category except category 1 cannot ignore the existence of processors in other categories which have higher priorities.

Nevertheless, we can apply equation 3.21 directly to processors in category 1:

$$n_1 f_1 \psi_1 = M[1-(1-\frac{f_1 \psi_1}{M})^{n_1}(1-\frac{1-(1-\frac{1-f_1}{M})^{n_1}}{M})^{M}] \qquad (3.23)$$

For the joint flow of processors in both category 1 and 2, the left-hand side of equation 3.21 becomes

$$n_1 f_1 \psi_1 + n_2 f_2 \psi_2 \ .$$

This is because in a conflict-free environment the total number of busy memory modules is just the sum of memory requests submitted from processors in both categories. On the other hand, the probability that a particular memory module is not requested by any of the non-queued processors in either category is

$$(1-\frac{f_1 \psi_1}{M})^{n_1}(1-\frac{f_2 \psi_2}{M})^{n_2} \ .$$

The total number of distinct memory modules occupied by queued processors in either category, r, is now (see equation 3.20)

$$r \approx M[1-(1-\frac{1-f_1}{M})^{n_1}(1-\frac{1-f_2}{M})^{n_2}] \qquad (3.24)$$

and, therefore, the probability that a queue makes a request (see section 3.3) is

$$[1-(1-\frac{1-f_1}{M})^{n_1}(1-\frac{1-f_2}{M})^{n_2}] \ .$$

The probability that a particular memory module is not busy because of requests submitted from queued processors thus becomes

$$[1- \frac{1-(1- \frac{1-f_1}{M})^{n_1}(1- \frac{1-f_2}{M})^{n_2}}{M}]^M .$$

Finally, combining all the above factors, the probability for a memory module to be busy serving a request from processors in either category is

$$1-(1- \frac{f_1\psi_1}{M})^{n_1}(1- \frac{f_2\psi_2}{M})^{n_2}[1- \frac{1-(1- \frac{1-f_1}{M})^{n_1}(1- \frac{1-f_2}{M})^{n_2}}{M}]^M .$$

By means of this probability we get the following bandwidth equation for the joint flow of processors in both category 1 and 2:

$$n_1f_1\psi_1 + n_2f_2\psi_2$$
$$= M[1-(1- \frac{f_1\psi_1}{M})^{n_1}(1- \frac{f_2\psi_2}{M})^{n_2}(1- \frac{1-(1- \frac{1-f_1}{M})^{n_1}(1- \frac{1-f_2}{M})^{n_2}}{M})^M] \qquad (3.25)$$

Solving for f_1 in equation 3.23, we can substitute its value into equation 3.25 to find f_2.

This process can then be repeated m times until we find f_m from

$$\sum_{i=1}^{m} n_i f_i \psi_i = M\left[1-\left(\prod_{i=1}^{m}\left(1-\frac{f_i\psi_i}{M}\right)^{n_i}\right)\left(1-\frac{1-\prod_{i=1}^{m}\left(1-\frac{1-f_i}{M}\right)^{n_i}}{M}\right)^{M}\right] . \qquad (3.26)$$

CHAPTER 4

Applications of the Memory Interference Models

4.1 Introduction

A good model of a phenomenon usually serves two purposes. On the one hand, it serves as a concise description of the phenomenon. It exposes the internal structure and relates the effects of various parameters involved in the phenomenon. Thus insights could be gained and useful predictions could be extrapolated from known results obtained through a limited number of experiments.

On the other hand, a model can be used to produce the same net effect as does the modeled phenomenon. The model of a lower-level phenomenon can be incorporated in a study performed on a higher level to take into account the effect of this lower-level phenomenon. This process can be viewed as the application of a model.

The derivations of various memory interference models presented in Chapters 2 and 3 have already achieved the first purpose above.

As examples of the applications of the various memory interference models previously derived, two cases are presented in the following. In section 4.2 we propose an algorithm for the run-time estimation of a program running in a multiprocessor system. Memory interference models are used to adjust the execution times of various jobs in the program whenever the number of processors available in the system and/or the amount of parallelism available in the program changes. A simulation study of the execution of matrix multiplication in a multiprocessor system with virtual memory is described in section 4.3. The size and complexity of the problem forbid a simulation below the page request level. Therefore, a memory interference model has to be employed again in the simulator to introduce the effect of memory interference on various timing data involved, and thus make the simulation results more realistic.

4.2 An Algorithm for the Run-Time Estimation of a Program in a Multiprocessor System

The execution time is one of the most important, if not the most important, performance indices of a computer program. The major purpose of multiprocessing, in which multiple processors are engaged in the execution of a single computation, is indeed to enhance this performance index.

However, the effectiveness of multiprocessing depends crucially on the way in which the computation is decomposed into various jobs. Improper decomposition usually introduces unnecessary precedence structures and/or unequal loads among processors, which often wastes available computing power by causing excessive processor waiting. (A case study of this issue on AMP-1 for a specific algorithm - the Gaussian Elimination is presented in the Appendix.) Since no clear criteria for good decomposition are available, adopting the decomposition with minimum execution time through comparison of execution times resulting from alternative decompositions becomes the best criterion available.

The execution time of a computation can, of course, be found by coding the program and then running it on the target machine. However, the high cost of software development prohibits random trials without some a priori ideas. An efficient method for the run-time estimation of a program is therefore desirable. This method could be used in the initial phase of a program design to determine a good computation decomposition.

Beizer[Bei70] proposed a Markov model for the purpose of analytically determining the execution time of a program. The model is a directed graph in which each arc is characterized by a triple (p,u,v), where p is the conditional probability that the flow of control, if it reaches the source vertex of the arc, will go through the arc; u is the mean execution time of the arc; and v is the variance of this time. Once the graph model is constructed, the estimation of execution time can be performed by the star-mesh transformation method [Fer78], which is adopted from electric circuit theory.

Unfortunately, Beizer's model was proposed for the conventional single-CPU machine. The steps in the graph model are performed sequentially.

We propose in the following an algorithm for the purpose of determining the execution time of a program running on a multiprocessor system. A system with a single CPU becomes a special case of the multiprocessor system for which the algorithm is designed.

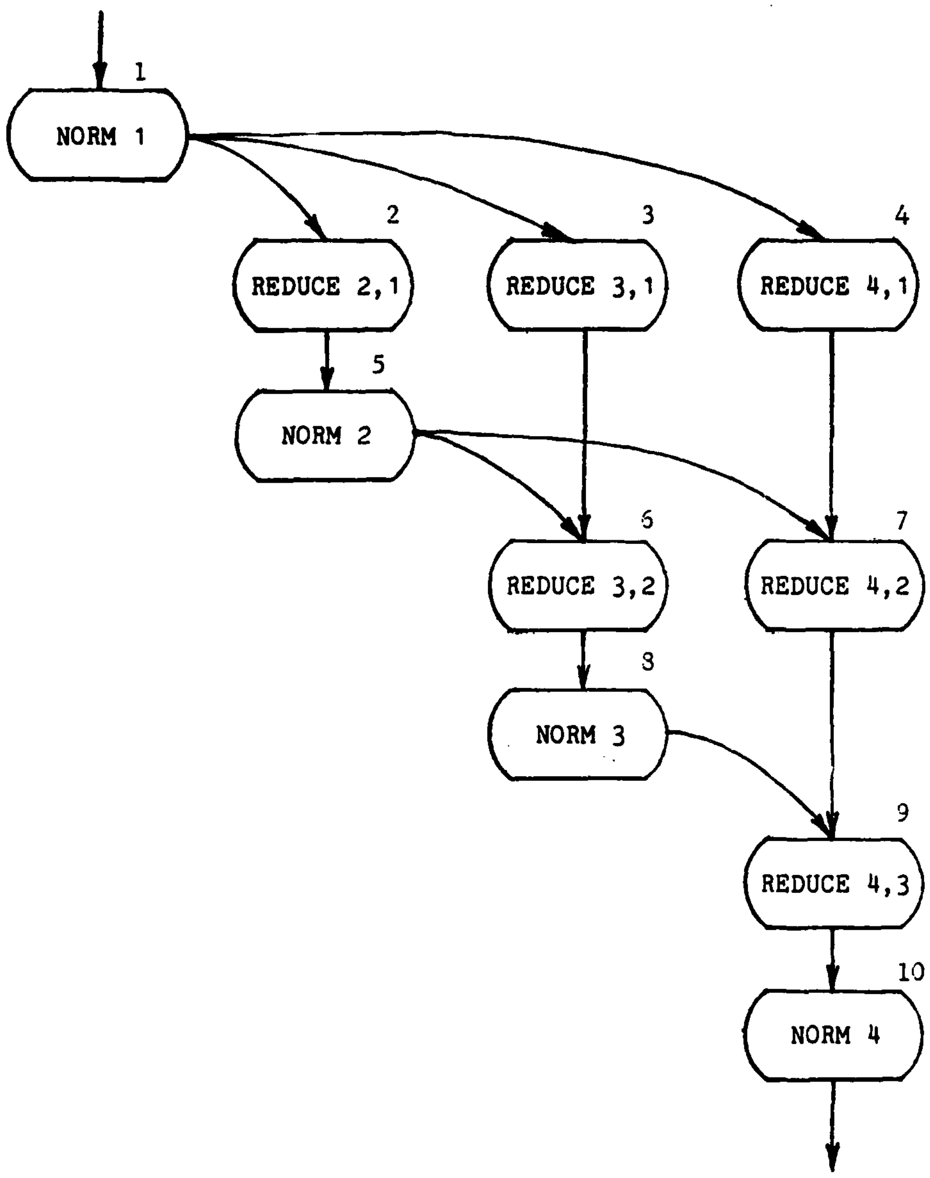
The approach presented here is based on the fact that computational processes can be modeled by graphs in which the vertices (nodes) represent single jobs and directed links represent the precedence relations. (Hence nodes take time to execute but links do not.) A job pointed to by a directed link can only start if the job at the source end of that link has been completed.

The graph (and thus the computation) can be represented in a computer by means of a Connectivity Matrix, C [Ram66], [RaG69a]. Let n be the total number of decomposed jobs in the precedence graph. C is then of dimension n x n such that C_{ij} is a "1" if and only if there is a directed link from node i to node j, and it is "0" otherwise.

Ramamoorthy and Gonzalez [RaG69b] proposed a connectivity-matrix-based technique for recognizing parallel processable streams in computer programs. In short, their technique iterates between locating as many all-zero columns (thus identifying nodes ready for processing) as possible and deleting from the connectivity matrix C these columns and the corresponding rows until no more columns or rows remain in C. By associating each job in the precedence graph (and thus each column in the connectivity matrix) with its job execution time and modifying the technique due to the finite number of processors available and the unequal execution times among jobs, their technique can be adapted to determine the execution time needed by a computation on a multiprocessor system. Furthermore, by employing one of the memory interference models presented previously depending on the environment, the job times can be dynamically adjusted to reflect the times needed for the execution more realistically.

For illustration, the precedence grar . of the triangularization phase of the Gaussian elimination algorithm (see Appendix) for a 4 x 4 matrix is shown in Figure 4.1. Also shown is its associated connectivity matrix. We shall assume there are no loops or strongly connected subgraphs [RaG69b] in the precedence graphs under consideration in order to insure the validity of our algorithm. (In fact if the precedence graph does contain strongly connected subgraphs, the maximal strongly connected subgraphs can be found either by inspection or by a simple analysis of its connectivity matrix [Ram66]. Each maximal strongly connected subgraph can then be considered as a single node, the computation decomposition appropriately redefined, and the resulting precedence graph will contain no strongly connected components or loops.) In addition, all the decomposed jobs in a precedence graph need to be executed in one run, and no job execution is preemptible. In cases where it is indeed necessary to specify data-dependent alternatives, we will deal with one possibility at a time and thus eliminate the data dependency of the precedence graph. The speed of the algorithm allows one to repeat the analysis easily for every possibility. A probability argument can be used to find out the mean execution time in these cases.

The paragraphs which follow describe an algorithm for determining the execution time of a program running on a multiprocessor system.



(a) Precedence Graph

Figure 4.1 Illustrations of the Run Time Estimation Algorithm

Col Row	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0
5	0	0	0	0	0	1	1	0	0	0
6	0	0	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0

(b) Connectivity Matrix

Figure 4.1 Illustrations of the Run Time Estimation Algorithm (continued)

Col	1	2	3	4	5	6	7	8	9	10
Job Time Row	5D	5S + 5M	5S + 5M	5S + 5M	4D	4S + 4M	4S + 4M	3D	3S + 3M	2D
1	0	1	1	1	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0
5	0	0	0	0	0	1	1	0	0	0
6	0	0	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0

(c) Augmented Connectivity Matrix

Figure 4.1 Illustrations of the Run Time Estimation Algorithm (continued)

(1) The first step is to represent the computation (or program) in terms of a precedence graph consisting of decomposed jobs as nodes and precedence relations as directed links (Figure 4.1(a)). Number all the nodes and represent this precedence graph by means of its associated connectivity matrix C (Figure 4.1(b)).

(2) Associate with each column of the connectivity matrix the expected (interference-free) job execution time of its corresponding job in the precedence graph (Figure 4.1(c)). These job execution times could be obtained for example by applying Beizer's method [Bei70] to the segments of sequential code corresponding to those jobs. In Figure 4.1(c) the job execution times are expressed, for simplicity, in terms of the number of floating-point operations required for the corresponding jobs. S, M and D in the job time expressions stand for the execution time of a floating-point subtraction, multiplication, and division, respectively. It should be emphasized, however, that the number of floating-point operations is used only to provide a relative scale for comparison. The execution time of a job certainly does not consist soly of floating-point operation times.

(3) Locate no more than P columns, if any exist, containing only zeroes in the connectivity matrix C, where P is the number of processors available in the system. Scheduling considerations could be involved at this point. Find the minimum among all job times in this set and subtract this minimum from each job time in the set. One processor will be assigned to execute each job in the set, and the execution proceeds until the job with the minimum job time is finished.

(4) Depending on the nature of the multiprocessor system used and the number of active processors participating in executing this set of jobs, we can get a value for the memory interference factor ρ of the system by employing one of the memory interference models developed in Chapters 2 and 3, whichever is appropriate. If no account of memory interference is desired, ρ will be set to 1.

(5) Multiply the memory interference factor ρ by the minimum job time found in (3), and then add the product to the total execution time elapsed so far.

(6) Delete from C both the columns and the rows corresponding to those jobs whose job times are now 0. Mark those jobs which were begun but not finished in this iteration, if any. These jobs will be assigned the highest priority for the selection process indicated in (3) over those newly-created all-zero columns because of the non-preemptibility assumption.

(7) Repeat steps (3) through (6) until no more columns or rows remain in the C matrix.

It can be shown that this procedure is valid for connectivity matrices of graphs which contain no loops [RaG69b].

For the decomposition given in Figure 4.1(c), the total execution times needed in a synchronous multiprocessor system with and without memory interference with various numbers of processors available in the system are shown in Table 4.1. The numbers enclosed in parentheses are speed-ups relative to the execution time using a single processor. For

Table 4.1

Execution Times and Speed-ups for the Problem in Figure 4.1

Using 4 Memory Modules

	no. of processors			
	1	2	3	4
memory interference ignored	144460ns (1.000)	93360ns (1.547)	93360ns (1.547)	93360ns (1.547)
synchronous multiprocessor system	144460ns (1.000)	100660ns (1.435)	100872ns (1.432)	100872ns (1.432)

convenience the execution times of floating-point operations in Figure 4.1(c) were substituted with values for the Digital Equipment Corporation's floating-point processor FP11-C used in the DEC PDP11/70 system [Dig76] (i.e. S=1130ns, M=2520ns, and D=3540ns). Again, the execution times of floating-point operations here are only used to provide a relative scale for comparison. We also assumed that there were four memory modules in the system, addresses were fully interleaved, and processors requested memory every cycle. A job whose precedence requirements are satisfied will be scheduled immediately whenever there is a processor available.

Note that a speed-up of only 1.435 is achieved, compared to an interference-free value of 1.547, when two processors are used. Further, it is interesting to note that the performance is not improved by using more than two processors. Actually it takes even more time to execute the problem when three or more processors are used. Although the additional memory interference is partly responsible for this anomaly, the major reason is probably the inadequacy of the scheduling strategy. It is clear from Figure 4.1(a) that more than 3 processors will not improve the performance, since there are at most 3 jobs that can be executed in parallel. However, it is possible to improve the performance with 3 processors by choosing a different scheduling strategy. It should be pointed out that there are no known efficient algorithms for scheduling a task system such as ours for shortest execution time [Cof76].

4.3 An Investigation of the Execution of Matrix Multiplication in a Multiprocessor System with Virtual Memory

In many applications of computers today, especially in scientific applications, huge matrices are frequently used to put data in a compact form, to reflect the topological structure of a problem, and so on. Linear-algebra operations are then used to manipulate these matrices. As more and more complicated problems and ambitious approaches are attempted, it is not unusual to find that the sizes of matrices involved grow beyond the size of available main memory in a system. In computers with virtual memories, these data matrices will generally be stored in the secondary storage and paged into main memory as needed.

As a typical example, an investigation of the execution of matrix multiplication in a multiprocessor system with virtual memory is presented in this section. Because of the lack of an existing multiprocessor virtual memory machine, the study was done by simulation.

For simplicity the multiprocessor computer system under investigation is assumed to be mono-programmed. In other words, whenever a page fault is encountered by a processor, that processor will wait until the page transfer is complete. One should notice, though, that in a multiprocessor system with virtual memory there exist two kinds of page faults. The first kind of page fault occurs when a processor attempts to access a page which is neither present in the main memory nor being loaded into the main memory by any other processor in the system. This kind of page fault will generate a page request to the secondary memory system and thus will be referred to hereinafter as an "effective page fault". On the other hand, it is possible for a processor to address a page which is being brought in by another processor but is not yet available for access. Since the loading process for that particular page has already been initiated, no request should be issued to the secondary memory system due to this fault. Nevertheless, the processor which encounters this kind of page fault still has to wait until the page transfer is complete before proceeding.

For economy the simulation was done at the page request level. A "global LRU page replacement scheme" was used in the simulation. That is, an LRU stack of depth M is kept in the system, where M is the number of page slots available in the main memory. The content in each level of the LRU stack is a page descriptor which contains a page number and the status of the corresponding page. Overall, the LRU stack consists of page descriptors corresponding to those pages which are assigned page slots in the main memory. These pages could, however, be in use by some processor, just present in the main memory, or in the process of being loaded into the main memory.

Since the simulation was not done at the individual memory request level, the order of page descriptors in the LRU stack at any instant did not exactly reflect the order of the latest accesses to recent pages. When a page is first accessed by a processor and thereafter brought into that processor's working set[Den70] (to be explained later), the corresponding page descriptor is updated and moved to the top of the stack. All the page descriptors originally above this descriptor, if any, are pushed downward a stack level. As long as the page stays in a processor's working set, its corresponding page descriptor just stays where it is in the LRU stack or gets pushed downward due to other new page accesses. When a page ceases to exist in any processor's working set, its corresponding page descriptor is removed from where it is, updated, and put below the lowest page descriptor corresponding to a page in use or being loaded by some processor. Hence all the pages being used or loaded will have their page descriptors maintained on the top portion of the LRU stack, followed by those descriptors corresponding to pages simply present in the main memory arranged in an LRU fashion.

Since the order of individual memory requests is not known, this approach does not work if the main memory is so small that thrashing [Den68] could occur. Since thrashing is not interesting to study anyway and not cost-effective even if memory is expensive, the size of the main memory in the simulation study was always kept large enough (for the execution of matrix multiplication using the block algorithm discussed below) to eliminate the possibility for thrashing to occur.

To be more specific, matrices A and B are to be multiplied together and their product is matrix C. In order to reduce the number of page faults, block paging [FiP79] was used for all three matrices involved in the multiplication. Furthermore, for convenience, the size of a page (and thus a data block) is assumed to be exactly equal to the capacity of a memory module. Treating page blocks as elements, each of the three matrices is of dimension N x N, where N is the number of blocks in a row or column.

The computation decomposition of the matrix multiplication proceeds as follows: The calculation of one product page block, a "super inner product", is treated as a basic job. This job actually consists of N submatrix multiplications for the N pairs of data page blocks. Initially all jobs are in a ready queue. Whenever a processor in the system becomes available, it is assigned a job by the job queue routine unless the queue is empty. Each processor halts when it finds the queue empty. (If processors cooperate on a super inner product instead, then whenever new data pages are needed all the processors will be idle for most of the page fault period. This is because the significant paging overhead is usually much larger than the individual job time when the computation is so decomposed. It is felt, therefore, it should be a better scheme to decompose the computation into larger pieces so that while one processor is waiting for a page transfer all the other processors can still be working. By properly scheduling the jobs, significant data page sharing can still be obtained.)

However, in order to further reduce the paging overhead, the least-recently-used feature of the page replacement scheme can be taken advantage of [Els74], [FiP79] to reduce the amount of paging by alternating the direction in which jobs are assigned (Figure 4.2). The order in which the N block multiplications within a job were performed was also alternated to take advantage of the LRU replacement scheme (Figure 4.2). More specifically, in terms of the page block multiplications, the way the jobs are assigned and the order the N block multiplications in a job are executed can be shown in the following algorithm:

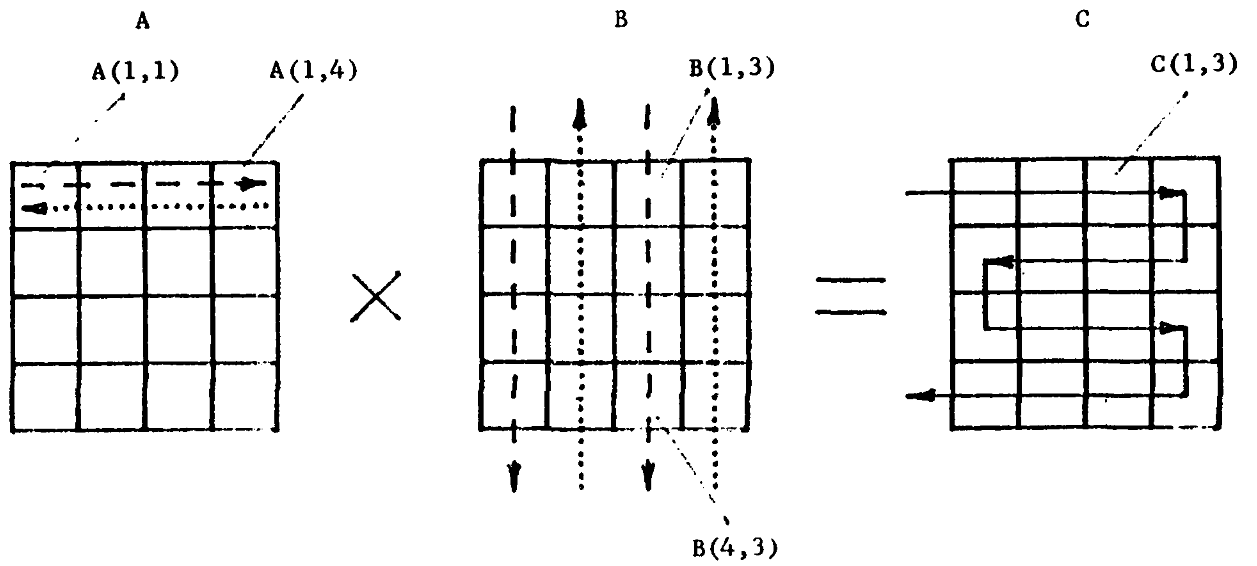


Figure 4.2 The Block Algorithm for Matrix Multiplication

```
FOR I := 1 STEP 1 UNTIL N DO

  BEGIN
    IF (I MOD 2) = 1
      THEN BEGIN JL:=1; JH:=N; JSTEP:=1 END
      ELSE BEGIN JL:=N; JH:=1; JSTEP:=-1 END;

    FOR J := JL STEP JSTEP UNTIL JH DO
      BEGIN
        IF (I+J MOD 2) = 0
          THEN BEGIN KL:=1; KH:=N; KSTEP:=1 END
          ELSE BEGIN KL:=N; KH:=1; KSTEP:=-1 END;

        C(I,J) := 0;                                  (4.1)

        FOR K := KL STEP KSTEP UNTIL KH DO
          C(I,J) := C(I,J) + A(I,K) x B(K,J)          (4.2)
      END
  END;
```

Note that the assignment, addition, and multiplication in statements 4.1 and 4.2 above are matrix assignment, matrix addition, and matrix multiplication, respectively. It is apparent that, except in the transition periods, each processor's working set contains 3 pages, one from each matrix.

The multiprocessor system studied consists of P identical arithmetic/logic processors, M identical memory modules, a crossbar interconnection network, and an I/O channel which takes care of page transfers between main memory and secondary storage. For the reasons mentioned in Chapter 3, the I/O channel has a higher priority in accessing the memory. Since the algorithm is simple, the corresponding program code, although addressed frequently, does not consume much memory space. Hence a local memory is assumed for each processor to accommodate its private copy of the program code. The processors thus only compete for the usually huge data storage. The request rate to the shared main memory is thus reduced and the performance of the system is improved.

A mutual-exclusion critical section is used in the multiprocessor program to allocate jobs to processors. It takes a finite amount of time, Tcs, for a processor to execute this critical section code. A processor could be locked out of this critical section for certain amount of time, Tlock, if some other processor is executing the critical section code when the processor attempts to access it.

A single first-in-first-out (FIFO) queueing facility is provided for processors waiting for the service of the I/O channel. A processor, therefore, will spend a finite time Tq in this queue before the I/O channel starts to serve it. The service time of the I/O channel for each page transfer basically consists of three time periods: the seek time Tseek, the latency time Tlat, and the page transfer time Tt.

The simulation was done using event-driven techniques and the relation among events and the time periods between events are shown in Figure 4.3. Note that there are two event lists in the simulation. Since the I/O channel has a higher priority for accessing the main memory, the page transfer is free from any interference. Furthermore, the seek time and latency time are characteristics only of the secondary memory system. The queueing time in the paging queue, therefore, is also interference-independent. However, all the other timing figures like the critical-section lock-out time Tlock, the critical-section execution time Tcs, and the computation time to multiply a pair of page blocks Tpair are affected by the memory interference.

The degree of memory interference itself depends at any given time on how many processors are competing for the main memory (not awaiting

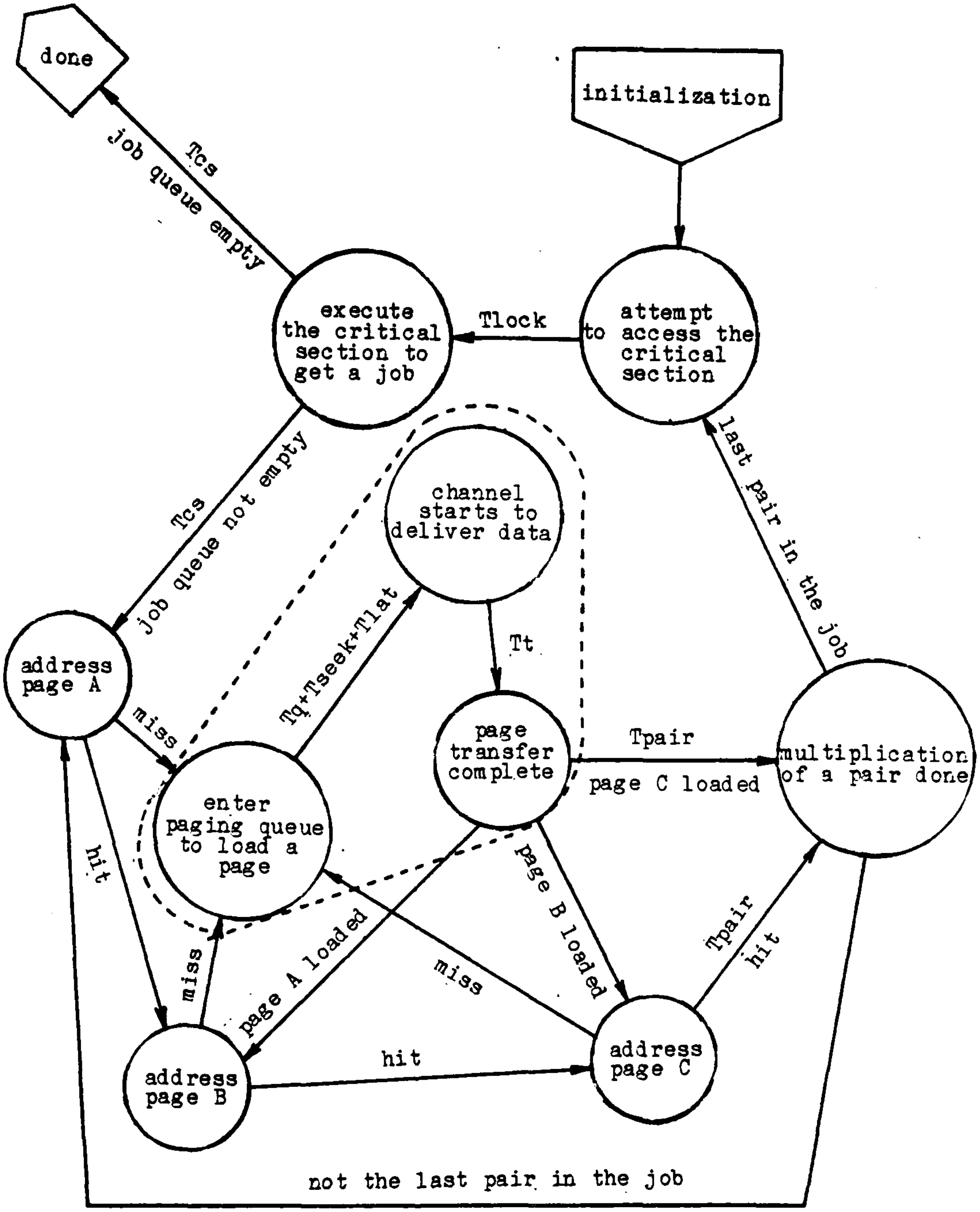


Figure 4.3 The Event-Driven Simulation

the page transfer, for example) and if the I/O channel is transfering data between main memory and the secondary storage. The memory interference factor, which is used as a multiplicative factor to modify those interference-dependent timing figures, has, therefore, to be evaluated from an appropriate memory interference model whenever the aforementioned conditions change. Because of this need for dynamically modifying some timing figures, two event lists were provided to save the cost of selecting modification targets and possibly reordering the occurrences of events in the lists to maintain their ascending order in time. One event list, called the event queue, was used to queue those events whose occurrences are interference-dependent and another event list, called the I/O queue, was used to queue those events whose occurrences are interference-independent. The event queue is used for activity outside the dotted line in Figure 4.3 and the I/O queue for activity inside.

Simulations were done for several different secondary storage systems. Various parameter values used in the simulation study are shown in Table 4.2. The processors' timing characteristics are compatible to the IBM System/370 series[CaP78], and the timing characteristics of the secondary storage system I are compatible to IBM 3330 disk storage unit [Hay78]. The secondary storage systems II, V, and X are just systems 2, 5, and 10 times faster than system I, respectively. The secondary storage system F, however, is identical to system I except a fixed head is used for each track to eliminate the seek time. Note that when we refer to a secondary storage system, we also include the I/O channel dealing with that system. The timing data enclosed in the brackets in

Table 4.2

Various Parameter Values Used in the Simulation Study

page size = 1K words
Tpair = 100ms [2000]
Tcs = 0.05ms [1]

processor's main memory request rate, $\psi_p = 0.25$

secondary storage system parameter	I	II	V	X	F
Tseek (ms)	30 [600]	15 [300]	6 [120]	3 [60]	0 [0]
Tlat (ms)	8.4 [168]	4.2 [84]	1.68 [33.6]	0.84 [16.8]	8.4 [168]
Tt (ms)	5 [100]	2.5 [50]	1 [20]	0.5 [10]	5 [100]
total overhead per page miss	43.4 [868]	21.7 [434]	8.68 [173.6]	4.34 [86.8]	13.4 [268]
channel transfer rate (x10^5 words/s)	2	4	10	20	2
channel request rate, ψ_c (when active)	0.1	0.2	0.5	1.0	0.1

Table 4.2 are normalized values with respect to the critical-section execution time Tcs. The model presented in section 3.2 Special Case II was used to evaluate the memory interference factor.

In all the simulations, two 64-page square data matrices were multiplied together to produce a third 64-page square matrix. Thus there were a total of 192 pages in the virtual address space. The main memory, however, has only 24 modules each capable of holding one page of data. The number 24 was chosen because for this particular computation even without page sharing the main memory is able to hold the working page sets of 8 processors, which is the maximum number of processors used in the simulation. The number of processors available in the multiprocessor system was varied from simulation to simulation to see the amount of speed-up achieved by increasing the number of available processors. This result is shown in Figure 4.4.

Apparently the secondary storage system I is too slow to support such a multiprocessor system with more than two processors for such a computation. System II, although only twice as fast, yielded much better performance. On the other hand, system V apparently matched the needs of the multiprocessor systems with up to 8 processors very well for such a computation. Hence doubling its speed did not improve the performance much. System F's high performance demonstrated the adverse effect of the dominating factor in the paging overhead - the seek time. Removing seek time here is almost as desirable as a five times overall speedup of the disk system. The upturn from 7 to 8 processors in Figure 4.4 (and also in Figure 4.5) is due to the fact that the number of decomposed jobs for

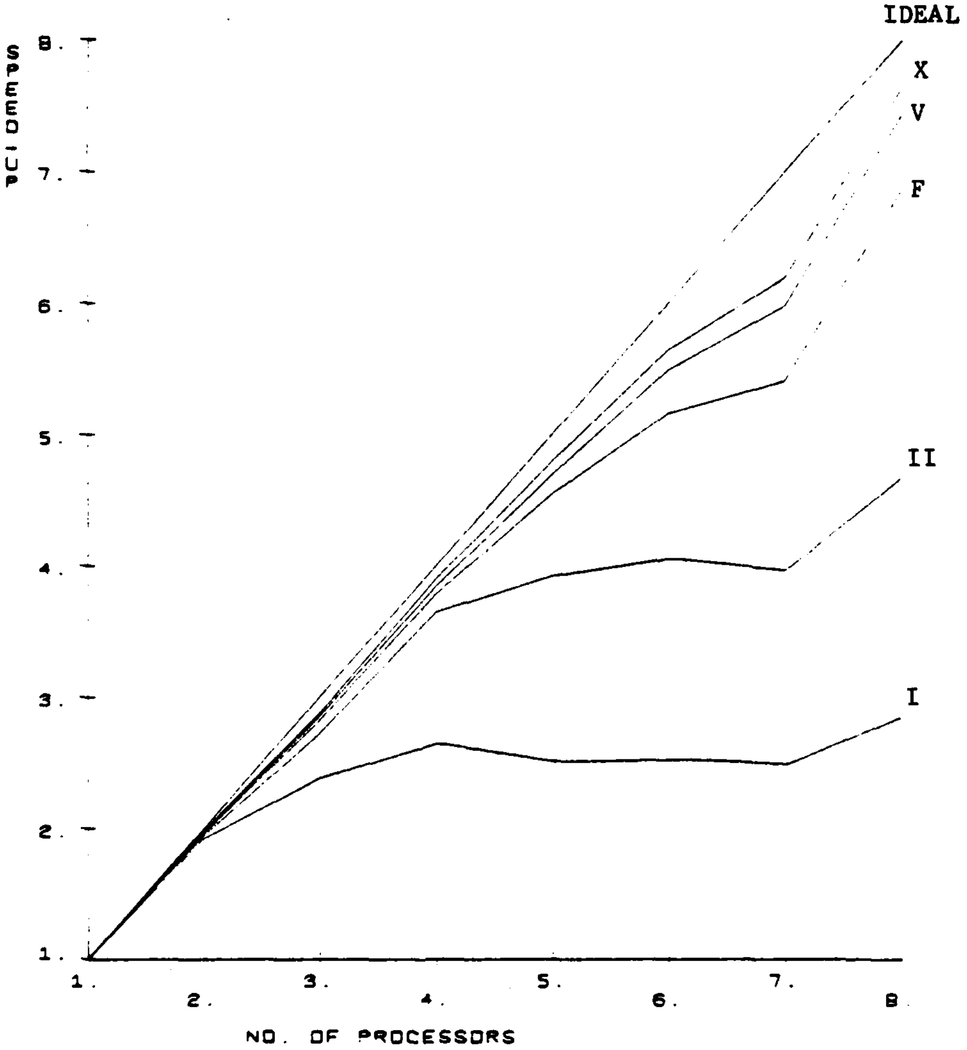


Figure 4.4 Speedups vs. No. of Processors

this computation is more evenly shared by 8 processors than by 7 processors (the end effect mentioned in section 2.8).

A few insights can be gained by observations of other parameters as functions of the number of processors used. For example, Figure 4.5 shows the average value of the memory interference factor, along the entire course of a simulation, versus the number of processors used for the computation. The resemblance of Figure 4.5 to 4.4 is probably not surprising. The performance of the multiprocessor system is directly related to the net computing power of the multiple processors in the system. Computing requires accesses to the shared main memory. Thus the performance of the system varies proportionally to the magnitude of the memory interference factor.

Figure 4.6 shows the ratio of the total number of effective page faults to the total number of all page faults. The shapes of the curves in the figure clearly point out the fact that the more seriously the secondary storage system "bottlenecks" the multiprocessor system, the larger the chance becomes for a processor to address a missing page whose loading procedure has been initiated.

Finally, since all the processors in the system participate in the execution of a known algorithm and the algorithm is so well designed that relatively long page-fault-free periods exist between clusters of page faults for any processor, it is interesting to look at the distribution of the intervals between page requests submitted to the secondary storage system by the multiple processors. Figure 4.7-4.9 are the histograms of such intervals for multiprocessor systems with secondary storage systems

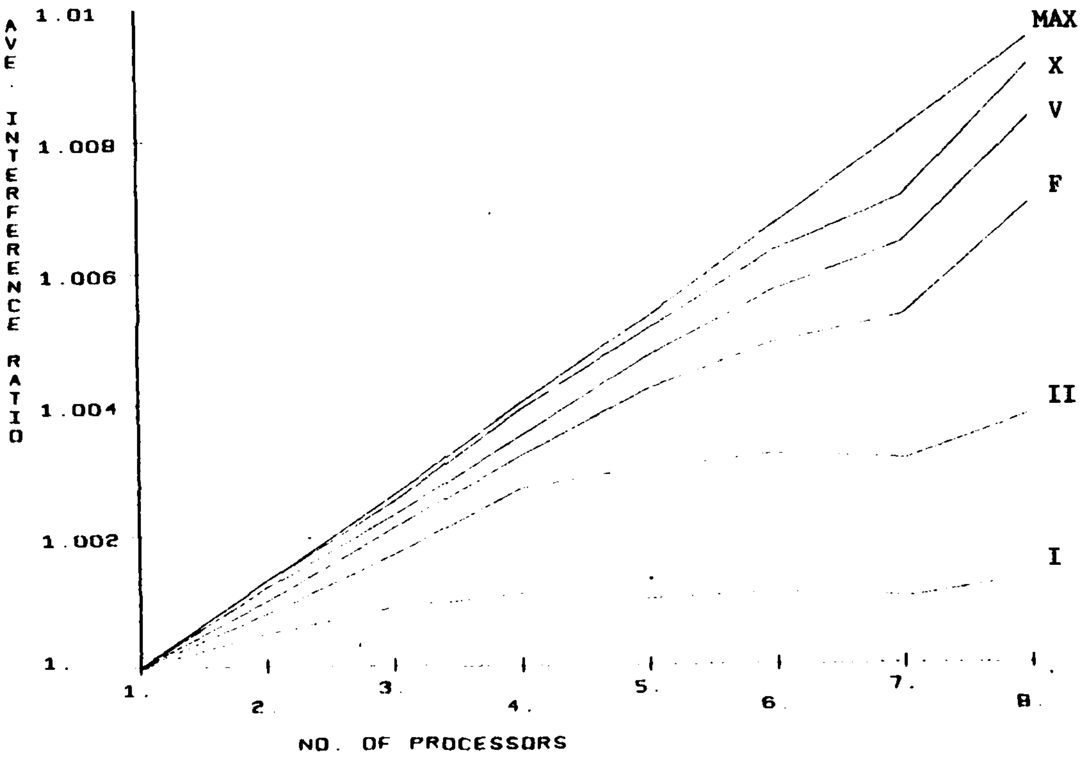


Figure 4.5 Average Value of the Memory Interference Factor vs. No. of Processors

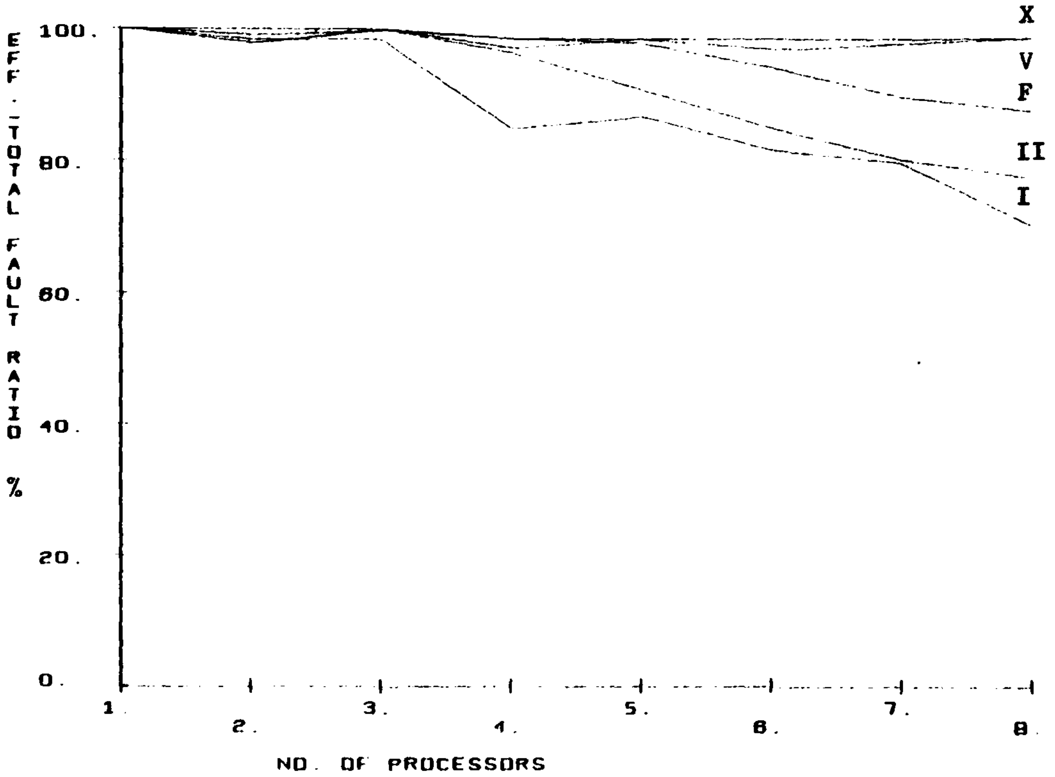


Figure 4.6 Effective-to-total Page Fault Ratio vs. No of Processors

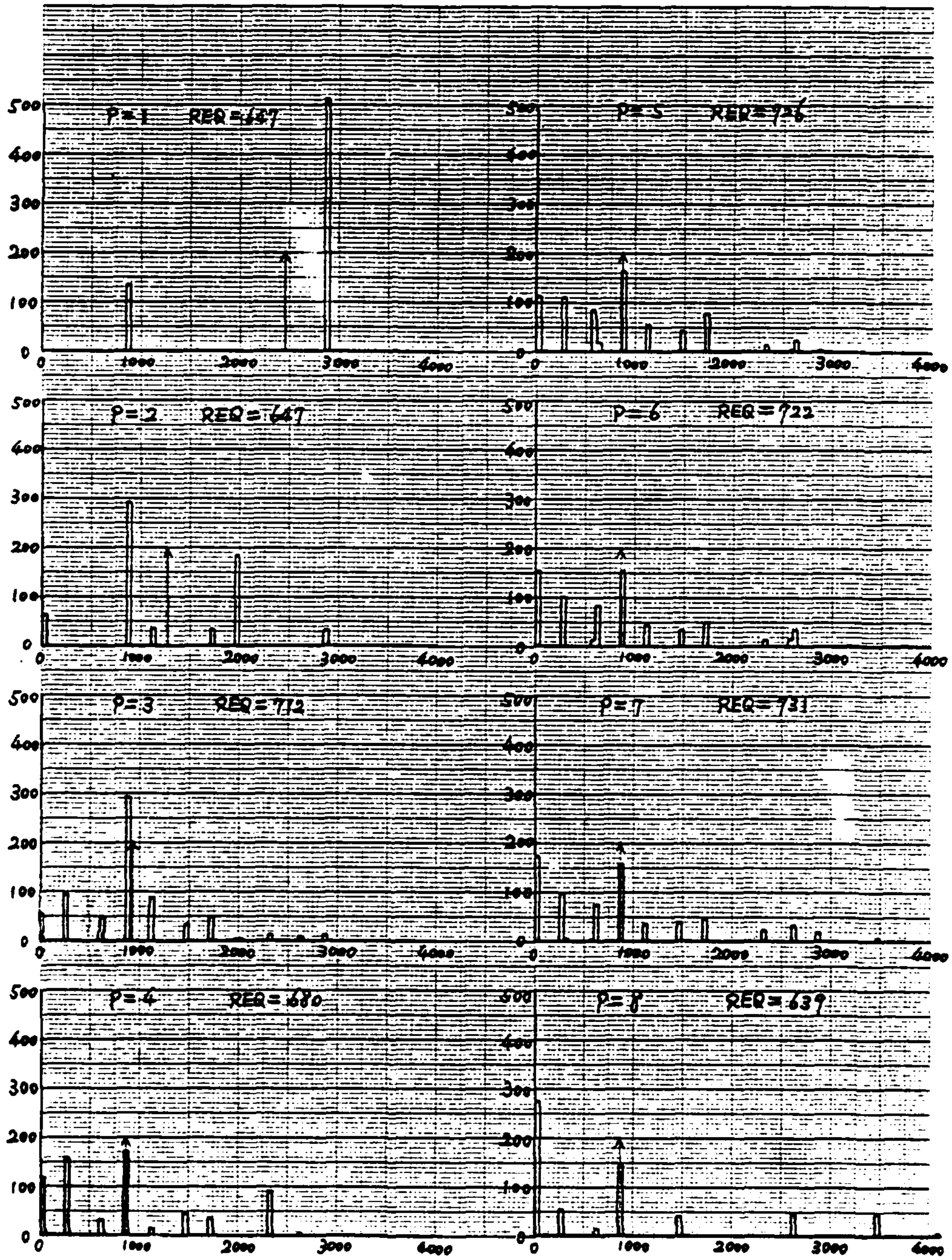


Figure 4.7 Histograms of Inter-page-request Intervals with Secondary Storage System I

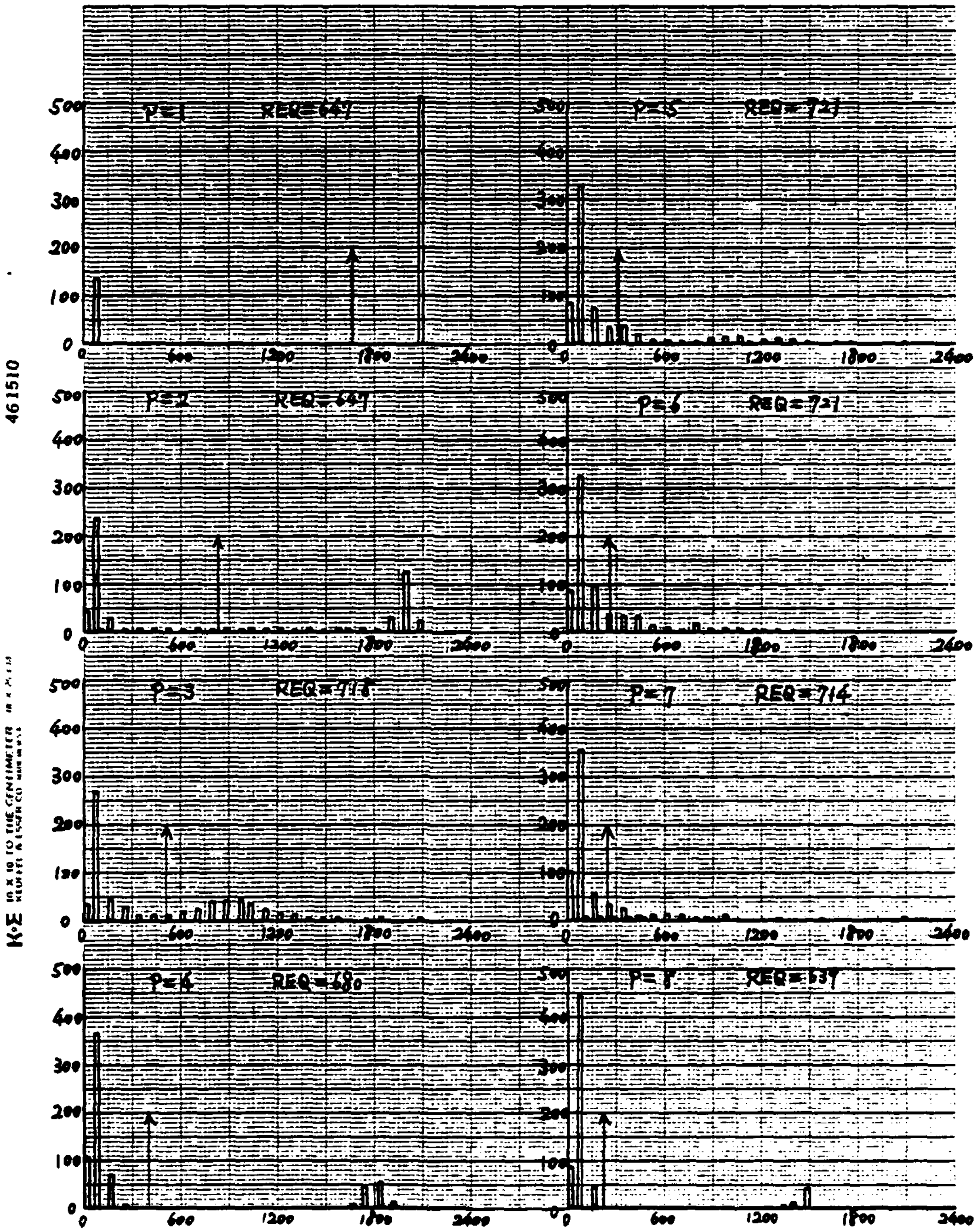


Figure 4.8 Histograms of Inter-page-request Intervals with Secondary Storage System X

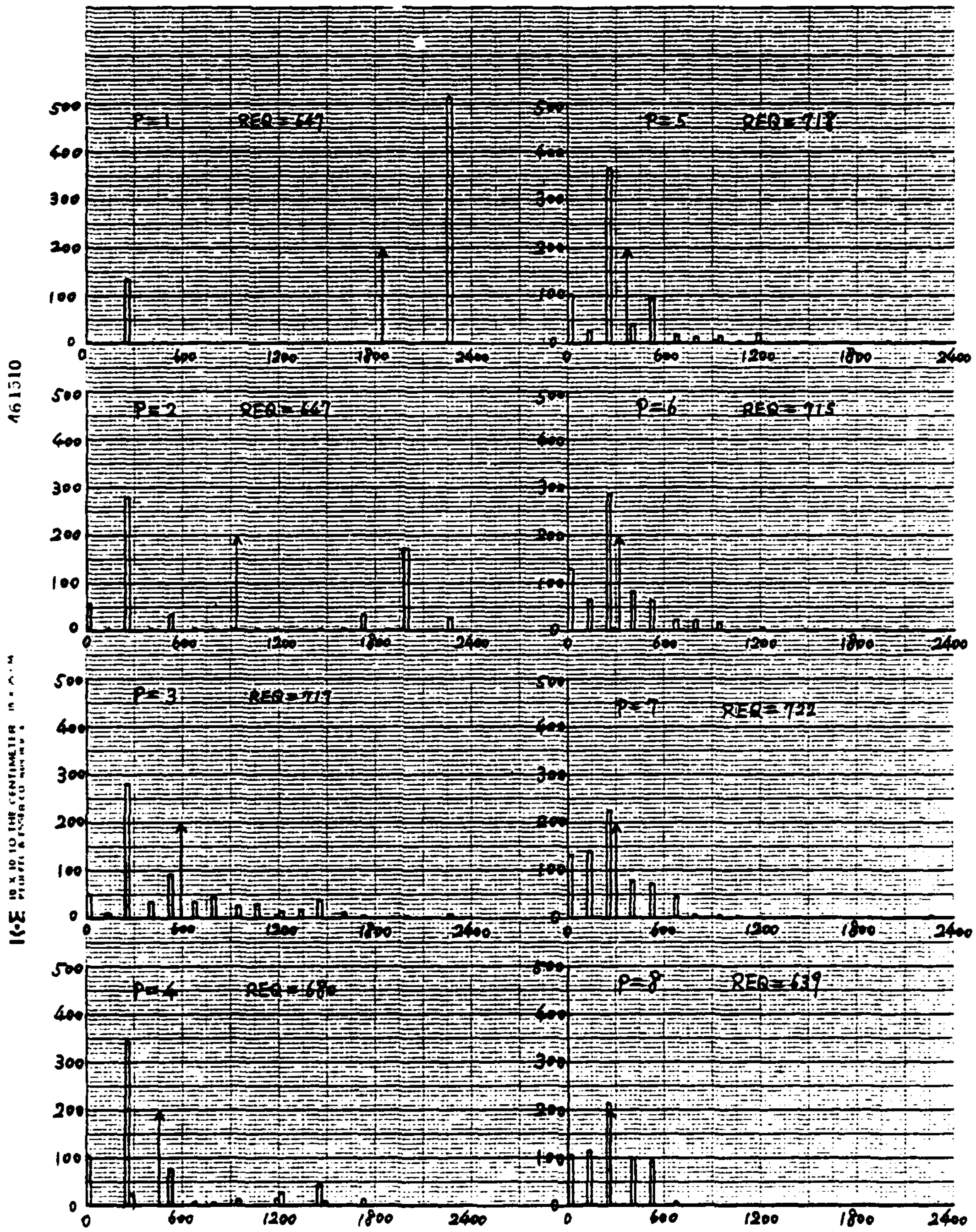


Figure 4.9 Histograms of Inter-page-request Intervals with Secondary Storage System F

I, X, and F, respectively. In each figure there is a histogram for each number of processors used (P). The total number of requests (REQ) is also indicated for each histogram. The horizontal axis is labelled by the normalized time (with respect to Tcs) and the vertical axis the absolute number of occurrences. The average value of the intervals for each histogram is indicated by a vertical arrow.

When only one processor is used for the matrix multiplication, there are only two values for the inter-request intervals. One value is the total overhead per page miss (see Table 4.2), which occurs when the processor is establishing a new working set, and the other value is the total overhead per page miss plus the execution time for the multiplication of a page block pair (Tpair), which occurs when a new working set is just established. However, as the number of processors in the system increases the lengths of the intervals between page requests mostly decrease and assume more values. On the other hand, for P = 6 in Figure 4.7 we do notice certain very long inter-request intervals. These intervals occur since when more processors are involved more distinct pages are brought into the memory. It is then possible for a processor to find its new working set already in the memory.

An interesting observation can be made from these histograms if we consider the secondary storage system as a server and the page requests as customers. The average length of inter-request time to the secondary storage can then be interpreted as the average interarrival time of customers to the server. This time in terms of the normalized time unit is indicated by the vertical arrow in each histogram. The total overhead per page miss given in Table 4.2 (which does not include any overhead for waiting in the channel queue) can be considered as the service time of the server for each customer. Notice that this service time is soon approached by the average interarrival time in Figure 4.7 when the number of processors increases. We recall from the queueing theory [Kle75] that an open random flow system (e.g. a M/M/1 queue) could become unstable when the customers' arrival rate approaches the server's service rate. It is actually the finite number of possible outstanding requests (because of the finite number of processors in the system) which keeps the system from going beyond saturation.

In contrast, although the arrival rate in Figure 4.8 for system X increases (the average interarrival interval decreases) as the number of processors increases, that rate never exceeds the maximal service rate of the server for up to eight processors. The fact that the system is not yet saturated actually guarantees further speed-up if even more processors are used.

Finally, since the average interarrival time has approached the service time of the server when eight processors are used in Figure 4.9 for system F, the speed-up curve in Figure 4.4 is expected to flatten out when more than eight processors are used in the multiprocessor system with secondary storage system F.

The above observation reveals information not contained in the speedup curves (Figure 4.4) and is helpful to system designers in balancing computer systems.

CHAPTER 5

Conclusions

5.1 Summary of Results

Multiprocessor systems can save hardware cost or afford better resources by sharing common resources among processors, but they pay for this saving by incurring access conflicts for resource usage. The idea here is cost-effectiveness, which means that the performance and the economy should be appropriately compromized. On the other hand, some resources in a multiprocessor system have to be shared to achieve synchronization. Thus performance degradation due to resource contention and access interference becomes inevitable in these systems. Because of the attractiveness of multiprocessing, it is, therefore, of great interest to understand such degradation in performance in order to minimize its effect by using available architectural parameters.

We have mainly focused our attention in this thesis on the special problem of memory interference in tightly coupled multiprocessor computer systems. Depending on the nature of the memory-requesting mechanism, discussion was centered on two important cases of such systems.

The memory interference in multiprocessor systems with time-division-multiplexed busses was first discussed in Chapter 2. The discussion started from Emer´s model for a multiple-instruction-stream pipelined processor with a single fixed-cycle shared resource, which was reviewed in section 1.2. Generalizations of that model for systems with multiple resources or resources with more general resource cycle times were discussed. Provisions for the application of the model to programs with critical sections treated as software resources were also covered. Furthermore, as presented in Chapter 2 as "the SCP problem", an effective resource cycle time was defined to replace the physical resource cycle time for a TDM-bus multiprocessor system with unassigned time slots. Finally, measured performance data from the execution of matrix multiplication on AMP-1 was used to check the above model. Matrix multiplication was chosen to focus on resource contention overhead only, since it has a large number of independent jobs with no precedence constraints among them. Two other models for matrix multiplication execution with and without modification by the memory interference factor were also presented for comparison. Model-predicted values by the hybrid model and the renewal-theory model with modification by the memory interference factor both yield errors which are less than 0.8% with up to 7 processors. These two models also model the imperfect job sharing at the end of the computation, which make them have the potential of dealing with precedence structures in general computations. the measured data.

In Chapter 3, attention was shifted to synchronous multiprocessor systems with crossbar as the interconnection network. A general probabilistic model with provisions for both non-uniform memory request rates and processor priorities was proposed. The model is superior to queueing-theory-based models in the fact that it can easily accomodate these provisions. For the case of uniform memory access, an improved model based on a steady flow concept was discussed. With the aid of simulation results, this model was compared to other models in the entire range of memory request rate ((0,1]) to demonstrate its accuracy. It was also shown that the model could be used iteratively to accomodate processor priorities.

Chapter 4 presents two application examples of the memory interference models. An algorithm for the estimation of the execution time of a program running on a multiprocessor system was proposed in section 4.2. Appropriate m~~ory interference model can be used to dynamically adjust the job execution times according to the actual number of processors active at any given time in the system. Section 4.3 presents another example, which is a case study on the execution of matrix multiplication in a multiprocessor system with virtual memory. A memory interference model was used to introduce the effect of memory interference into the simulation study, which was done on the page request level.

5.2 Suggestions for Future Research

The effort in section 2.5 is intended to deal with the performance overhead caused by critical section code in multiprocessor programs. Although a mutual-exclusion critical sectior code is indeed a shared resource in a multiprocessor system, the probabilistic approach used in section 2.5 may not work well if the critical section is not frequently accessed or the critical section takes too much time to execute. The assumption of independence among requests could be poor in the latter case because of potentially serious congestion. Further research is thus required to explore more deeply more precise approaches to assess this overhead.

Secondly, the importance of computation decomposition for a multiprocessor algorithm is briefly mentioned in several places in the thesis. However, not much investigation of this issue is found in the literature and no good general guideline for decomposing a computation for multiprocessing is available. The algorithm proposed in section 4.2 is the first step toward attacking this problem systematically, but it does not work in all cases. For example, certain shared data in a multiprocessor program may be protected by a mutual-exclusion critical section to insure its integrity. Any job which needs access to this kind of critical section may be blocked temporarily by other jobs, but no definite precedence relation exists. Since the performance degradation of a multiprocessor system due to avoidable precedence relations in a multitasked computation may very well exceed the overhead caused by memory access interference, more research on the computation decomposition is definitely necessary. The renewal-theory model presented in section 2.8 may have the potential of dealing with the precedence structures of general computations.

Finally, since a cache is effective in enhancing the performance of the memory system and greatly reduces the traffic between processors and the main memory, it has been widely adopted in conventional uniprocessor systems. However, the implementation of a cache or caches in a multiprocessor system has its unique problems. The multiple-copy problem occurs if a separate private cache is provided for each processor, while serious interference could occur if the cache is shared among processors. In view of all the potential advantages of using caches in computer systems, this issue definitely deserves more research.

REFERENCES

[BaS76] Baskett, F., and A.J. Smith, "Interference in Multiprocessor Computer Systems with Interleaved Memory," *Communications of the ACM*, vol. 19, no. 6, pp.327-334, June 1976.

[Bei70] Beizer, B., "Analytical Techniques for the Statistical Evaluation of Program Running Time," *AFIPS Conference Proceedings* 37 (FJCC), pp.519-524, 1970.

[Bha75] Bhandarkar, D.P., "Analysis of Memory Interference in Multiprocessors," *IEEE Transactions on Computers*, vol. C-24, pp.897-908, September 1975.

[BrDa77] Briggs, F.A., and E.S. Davidson, "Organization of Semiconductor Memories for Parallel-Pipelined Processors," *IEEE Transactions on Computers*, Vol. C-26, pp.162-169, February 1977.

[BrDe71] Brown, K.M., and J.E. Dennis, Jr., "On the Second Order Convergence of Brown´s Derivative-free Method for Solving Simultaneous Nonlinear Equations," Yale University Department of Computer Science Technical Report, pp.71-77, 1971.

[Bri77] Briggs, F.A., "Memory Organizations and Their Effectiveness for Multiprocessing Computers," CSL Report R-768, University of Illinois, May 1977.

[Bro69] Brown, K.M., "A Quadratically Convergent Newton-like Method Based upon Gaussian Elimination," *SIAM Journal on Numerical Analysis*, pp.560-569, 6(4)1969.

[BuK71] Budnik, P.P., and D.J. Kuck, "The Organization and Use of Parallel Memories," *IEEE Transactions on Computers*, vol.20, pp.1566-1569, 1971.

[CaP78] Case, R.P., and A. Padegs, "Architecture of the IBM System/370," *Communications of the ACM*, vol.21, no.1, pp.73-95, January 1978.

[CES71] Coffman, E.G., Jr., M.J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys*, 3, no.2, pp.67-78, June 1971.

[Che71] Chen, T.C., "Parallelism, Pipelining, and Computer Efficiency," *Computer Design*, pp.69-74, 1971.

[CKL77] Chang, D.Y., D.J. Kuck, and D.H. Lawrie, "On the Effective Bandwidth of Parallel Memories," *IEEE Transactions on Computers*, pp.480-489, May 1977.

[Cof76] Coffman, E.G., *Computer and Job/Shop Scheduling Theory*, John Wiley & Sons, Inc. New York, 1976.

[Cox62] Cox, D.R., *Renewal Theory*, John Wiley & Sons Inc., 1962.

[Dav80] Davidson, E.S., "A Multiple Stream Microprocessor Prototype System : AMP-1," Proceedings of the 7th Annual Symposium on Computer Architecture, vol.8, no.3, pp.9-16, May, 1980.

[Den68] Denning, P.J., "Thrashing: Its Causes and Prevention," Proceedings, AFIPS 1968 Fall Joint Computer Conference, vol.33, pp.915-922, 1968.

[Den70] Denning, P.J., "Virtual Memory," Computing Surveys, vol.2, no.3, September 1970.

[Dig76] Digital Equipment Corporation, PDP11/70 Processor Handbook, 1976.

[Els74] Elshoff, J.L., "Some Programming Techniques for Processing Multi-dimensional Matrices in a Paging Environment," National Computer Conference, pp.185-193, 1974.

[Eme79] Emer, J.S., "Shared Resources for Multiple Instruction Stream Pipelined Processors," CSL Report R-838, University of Illinois, July 1979.

[Fer78] Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, Inc. 1978.

[FiP79] Fischer, P.C., and R.L. Probert, "Storage Reorganization Techniques for Matrix Computation in a Paging Environment," Communications of the ACM, vol.22, no.7, pp.405-415, July 1979.

[FuO76] Fuller, S.H., and P. Oleinick, "Initial Measurements of Parallel Programs on a Multi-mini-processor," Proceedings, Computer Conference, pp.358-363, 1976.

[Hay78] Hayes, J.P., Computer Architecture and Organization, McGraw-Hill Book Company, p.344, 1978.

[Hoo77] Hoogendoorn, C.H., "A General Model for Memory Interference in Multiprocessors," IEEE Transactions on Computers, vol.C-26, no.10, pp.998-1005, October 1977.

[HoR77] Hon, R.W., and D.R. Reddy, "The Effect of Computer Architecture on Algorithm Decomposition and Performance," Proceedings of the Symposium on High Speed Computer and Algorithm Organization, University of Illinois, Urbana, Illinois, pp.411-422, April, 1977.

[Hor78] Horst, R.W., "The Design and Implementation of a Synchronous Multiple Microprocessor System: Part I," M.S. thesis, Department of Electrical Engineering, University of Illinois, Urbana, Illinois, 1978.

[IsM80] Isloor, S.S., and T.A. Marsland, "The Deadlock Problem: An Overview," Computer, vol.13, no.9, pp.58-78, September 1980.

[KaD79] Kaminsky, W.J., and E.S. Davidson, "Developing a Multiple-Instruction-Stream Single-Chip Processor," Computer, pp.66-76, December 1979.

[Kle75] Kleinrock, L., Queueing Systems, Volume 1: Theory, John Wiley & Sons, Inc., 1975.

[Kra77] Kravitz,.R.H., "The Design and Implementation of a Synchronous Multiple Microprocessor System: Part II," M.S. thesis, Department of Electrical Engineering, University of Illinois, Urbana, Illinois, 1977.

[Kun76] Kung, H.T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," Algorithms and Complexity: New Directions and Recent Results, J.F. Traub, Ed., Academic Press, New York, 1976.

[Law73] Lawrie, D.H., "Memory-Processor Connection Networks," Ph.D. thesis, Department of Computer Science, Technical Report 73-557, University of Illinois, Urbana, Illinois, February 1973.

[Law75] Lawrie, D.H., "Access and Alignment of Data in an Array Processor," IEEE Transactions on Computers, vol.C-24, no.12, pp.1145-1155, December 1975.

[Leh66] Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors," Proceedings of the IEEE, vol. 54, pp.1889-1901, December 1966.

[OlF78] Oleinick, P.N., and S.H. Fuller, "The Implementation and Evaluation of a Parallel Algorithm on C.mmp," Technical Report CMU-CS-78-125, Department of Computer Science, Carnegie-Mellon University, June 6, 1978.

[Pat79] Patel, J.H., "Processor-Memory Interconnections for Multiprocessors," Conference Proceedings of 6th Annual Symposium on Computer Architecture, Philadelphia, Pa., pp.168-177, April 1979.

[Pir67] Pirtle, M., "Intercommunication of Processors and Memory," Fall Joint Computer Conference, pp.621-633, 1967.

[RaG69a] Ramamoorthy, C.V., and M.J. Gonzalez, "Recognition and Representation of Parallel Processable Streams in Computer Programs - II (Task/Process Parallelism)," 1969 National ACM Conference, 1969.

[RaG69b] Ramamoorthy, C.V., and M.J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," Fall Joint Computer Conference, pp.1-15, 1969.

[Ram66] Ramamoorthy, C.V., "Analysis of Graphs by Connectivity Considerations," Journal of ACM, vol.13, no.2, pp.211-222, April 1966.

[Rau77] Rau, B.R., "Program Behavior and the Performance of Memory Systems," Ph.D. dissertation, Stanford University, Stanford, CA, July 1977.

[Rau79] Rau, B.R., "Interleaved Memory Bandwidth in a Model of a Multiprocessor Computer System," IEEE Transactions on Computers, vol. C-28, pp.678-681, September 1979.

[Rav72] Ravi, C.V., "On the Bandwidth and Interference in Interleaved Memory Systems," IEEE Transactions on Computers, vol. C-21, pp.899-901, August 1972.

[Smi77] Smith, A.J., "Multiprocessor Memory Organization and Memory Interference," Communications of the ACM, vol.20, no.10, pp.754-761, October 1977.

[Str70] Strecker, W.D., "Analysis of the Instruction Execution Rate in Certain Computer Structures," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1970.

[Str79] Strecker, W.D., "An Analysis of Central Processor - Input-Output Processor Contention," 1979 Conference on Simulation, Measurement and Modeling of Computer Systems, pp.27-40, August 1979.

[WuB72] Wulf, W.A., and C.G. Bell, "C.mmp - A Multi-Mini-Processor," Fall Joint Computer Conference, pp.765-777, 1972.

APPENDIX

The Effect of Computation Decomposition on the Performance of Executing Gaussian Elimination on AMP-1

Gaussian Elimination is a well-known classical algorithm for solving simultaneous linear equations. In terms of the matrix form Ax = b, the algorithm proceeds by triangularizing the A matrix first and then solving for the unknowns by backward substitution.

A Gaussian Elimination program was developed for the AMP-1 to solve a set of 14 linear equations of the form Ax = b[Dav80]. A 5-byte floating-point format was used for each matrix element. This format provides for an 8-bit exponent and a 32-bit mantissa which allows numerical precision comparable to most large computers.

Three versions of the program were written for performance evaluation. GAUSB decomposes the computation into three kinds of jobs: normalization, NORM i, which normalizes row i of the A matrix and b_i using A_{ii} (assuming $A_{ii} \neq 0$); reduction, REDUCE i,j ($j<i$), which subtracts the product of A_{ij} and row j from row i of A and $A_{ij}b_j$ from b_i to make the new $A_{ij}=0$; and back substitution, BKSUB i,j ($j<i$), which subtracts the product $A_{ji}b_i$ from b_j. Job precedence for normalization jobs and reduction jobs is shown in Figure A.1 and that for back substitution jobs is shown in Figure A.2. No back substitution jobs can proceed until all the normalization and reduction jobs are done. Job precedence is controlled in GAUSB by the job-allocating critical section.

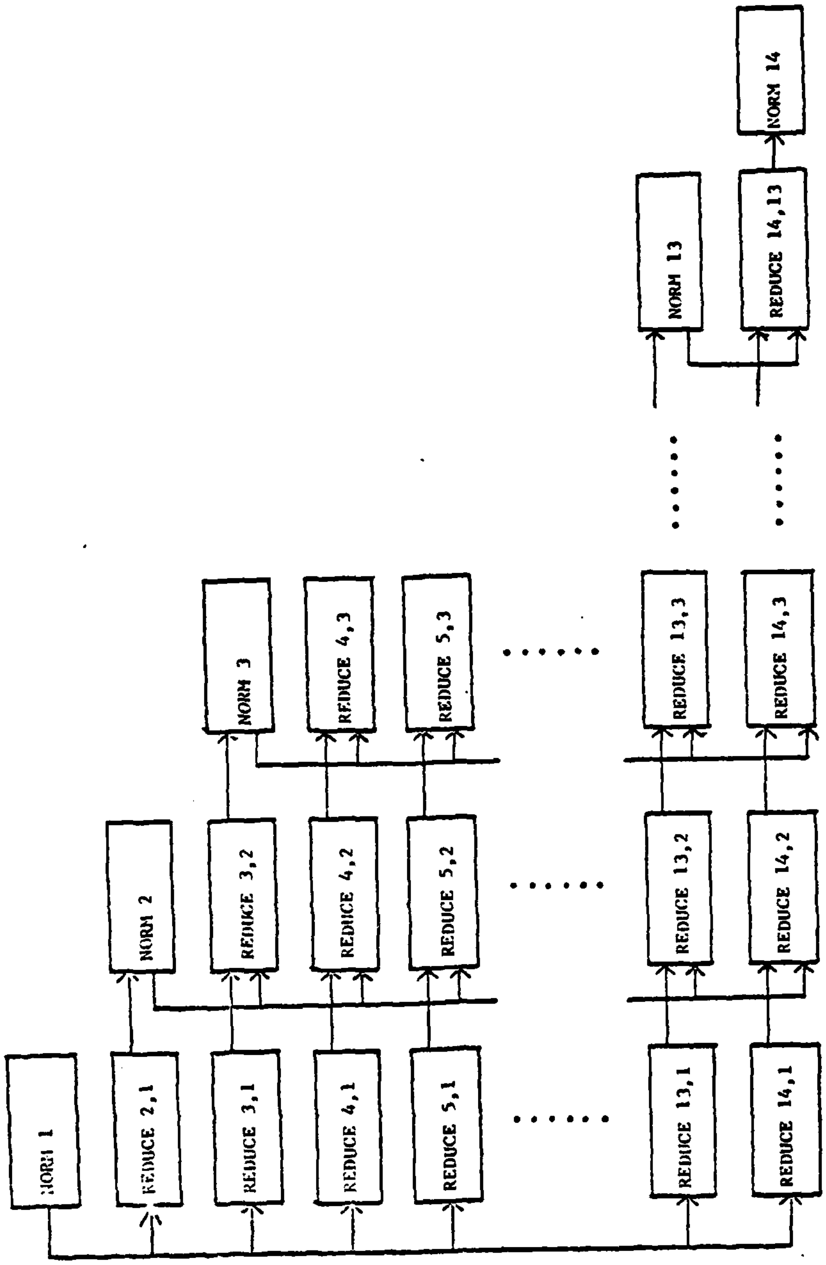


Figure A.1 Job Precedence for Normalization Jobs and Reduction Jobs

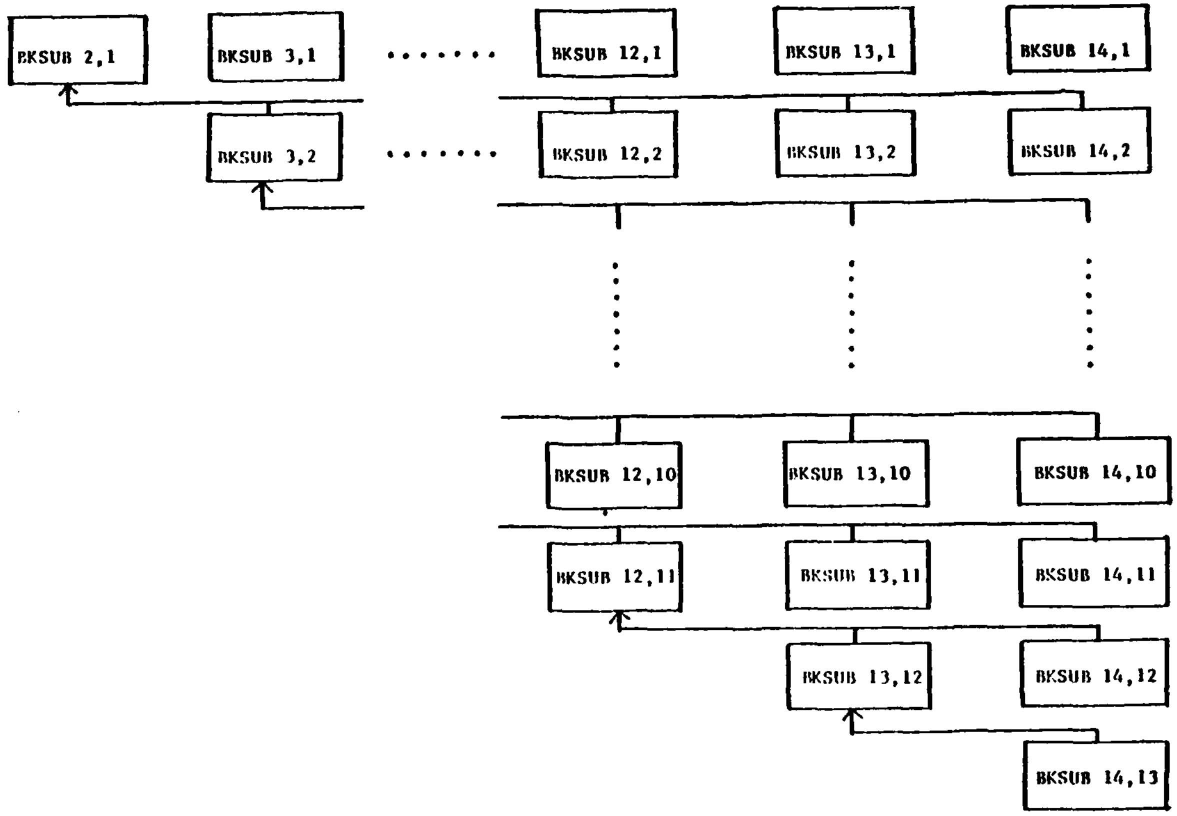


Figure A.2 Job Precedence for Back Substitution Jobs

GAUSY uses a separate semaphore for each of the decomposed jobs to reduce semaphore congestion. GAUSZ eliminates all normalization jobs by distributing the normalizing divides into the reduction jobs and back substitution jobs to reduce job precedence wait. All three versions require exactly the same number of each type of floating-point operation.

Since the performance of GAUSY is close to GAUSB in Figure A.3, it is seen that semaphore congestion is minimal. However, the normalization jobs in GAUSB did cause significant job precedence wait as the number of processors increased, as evidenced by the performance improvement of GAUSZ over GAUSB. These data illustrate the utility of reordering and revising the computations of standard algorithms developed for single processors when multitasking these algorithms, by proper decomposition, for multiprocessing.

The values for I in Figure A.3 indicate the number of ways the addresses are interleaved among the 64 memory modules in the system. The program code and data span at least 8 modules when I=1 and all 64 when I=64.

Some experiments concerning the effect of address interleaving on the performance were performed for GAUSZ, the version with the best performance as far as job precedence constraints are concerned. Figure A.4 displays the result. It should be noted that in these experiments, the number of memory modules was always 64, regardless of the degree of interleaving. Thus the performance for no interleaving or a low degree of interleaving as indicated is higher than one would expect if the number of modules were reduced to be equal to the degree of interleaving.

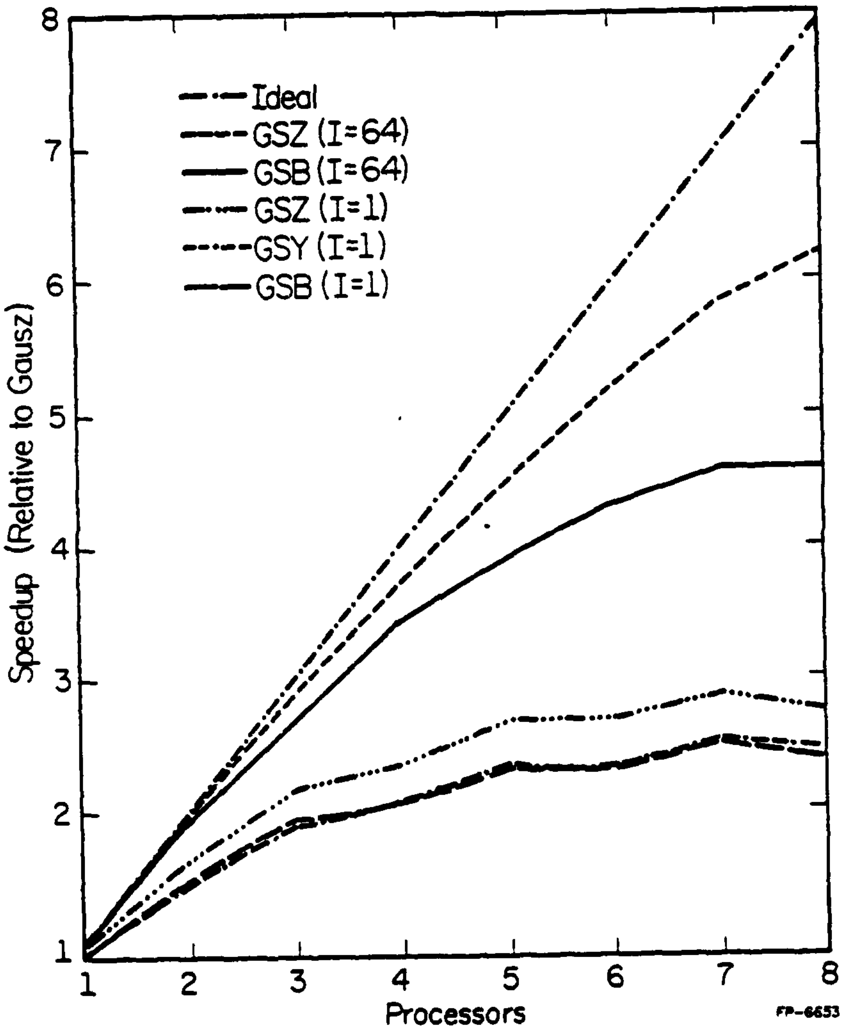


Figure A.3 Gaussian Elimination Performance

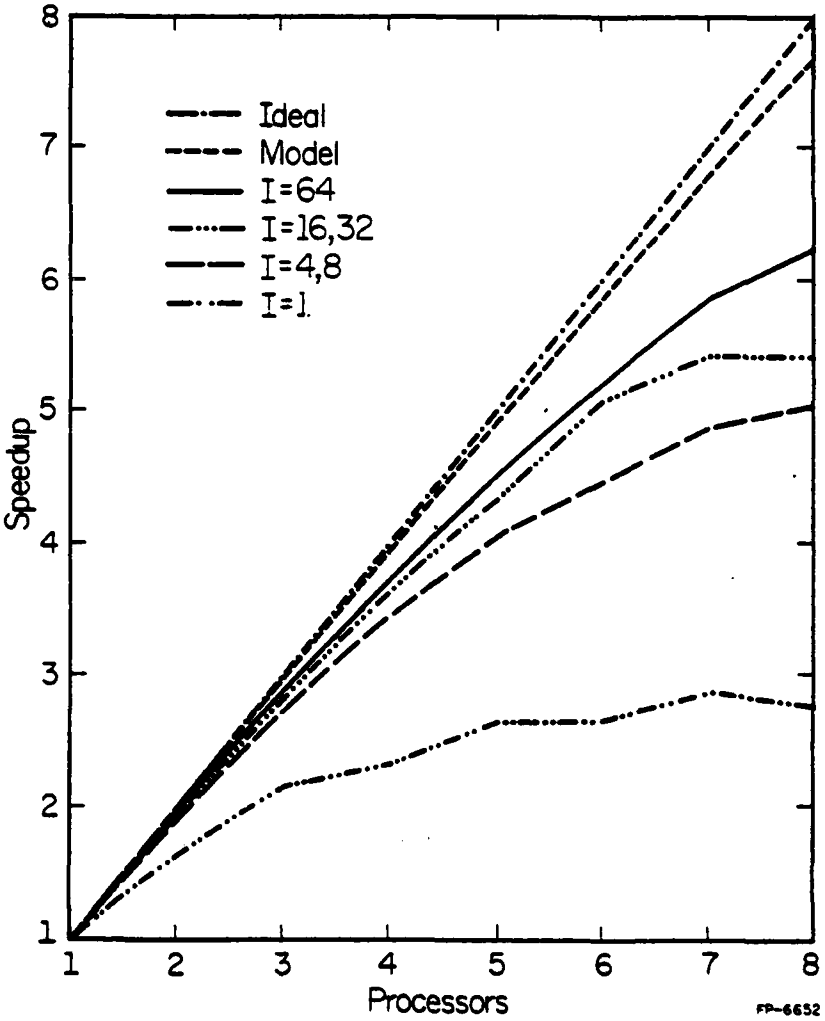


Figure A.4 GAUSZ Performance

It is interesting to note that even though so many modules of memory are used with respect to the number of processors a high degree of memory address interleaving among these modules is also required for high performance. Performance improvements are still obtainable by increasing I from 32 to 64 even when only 8 processors are used. The "Model" curve in Figure A.4 indicates the expected speedup if memory access contention were the only degradation from "Ideal" with I = 64. In fact the measured memory access contention is indistinguishable from the Model curve in Figure A.4. It may be inferred from these data that memory access contention is negligible when I = 64 and that job precedence and other degradation factors account for the major differences between actual and ideal speedup in this region. Memory access contention becomes more significant when I is reduced.

For comparison, similar experiments were done for the matrix multiplication program MXMC, and the result is shown in Figure A.5. Again, experiments always use 64 memory modules, regardless of I. High degrees of interleaving result in a performance which comes remarkably close to the ideal, despite the memory access contention due to shared code as well as shared data space for the matrices. Also, the fairly straight speedup curves indicate the lack of precedence constraints in the computation.

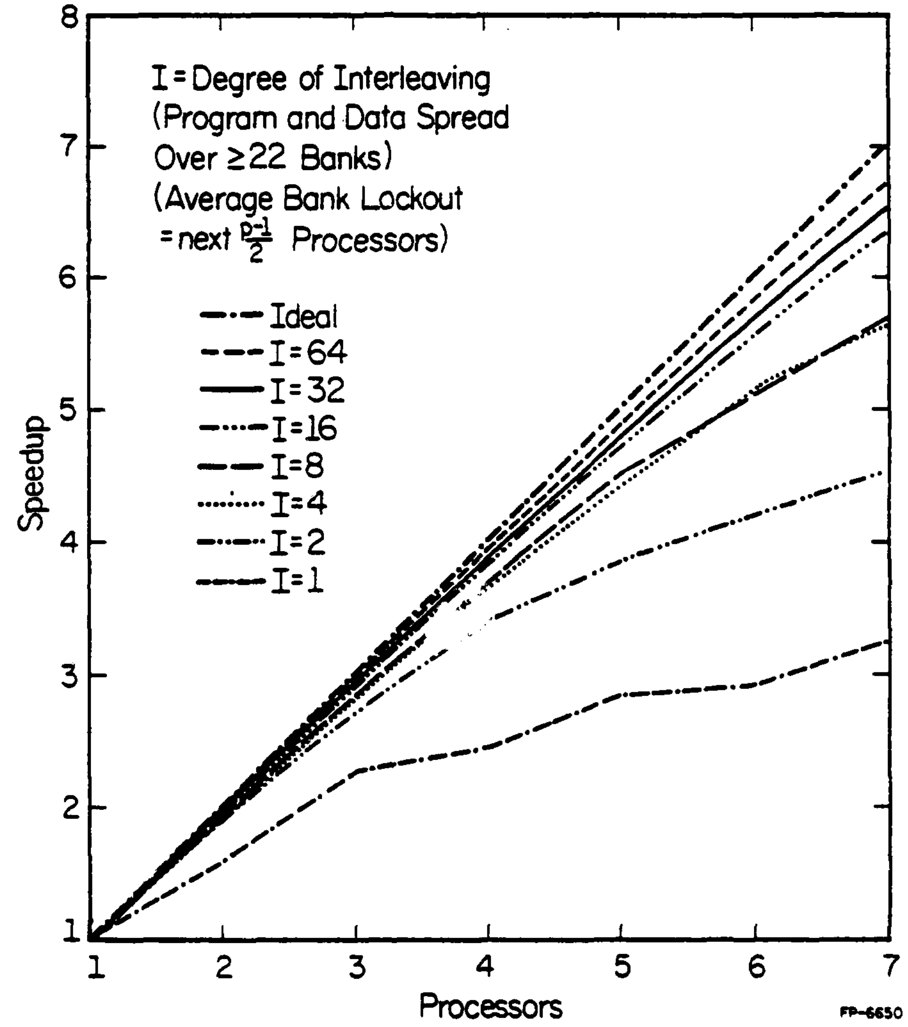


Figure A.5 MXMC Performance

VITA

David Wei-Luen Yen was born in Taiwan, Republic of China on September 24, 1951. He received the B. S. degree in Electrical Engineering from National Taiwan University, Taiwan, China, in 1973 and the M. S. degree in Electrical Engineering from the University of Illinois in 1977. At the University of Illinois he was employed as a teaching assistant in the Department of Electrical Engineering working on PLATO programming in the fall of 1975, a research assistant in the Control Systems Research Laboratory from 1976 to 1977, a teaching assistant in the Microcomputer Laboratory of the Electrical Engineering Department in 1978, and a research assistant at the Coordinated Science Laboratory from 1979 to 1980.

ENd